(2)

Final Technical Report

3D Navier–Stokes Flow Analysis for a Large–Array Multiprocessor

Sponsored by
Defense Advanced Projects Agency (DoD)
Defense Small Business Innovation Research Program
3D Navier–Stokes Flow Analysis for Large Array Multiprocessor
ARPA Order No. 5916, Amdt. 9
Issued by U.S. Army Missile Command Under
Contract # DAAH01-88-C-0405

AMTEC ENGINEERING, INC.
3055 112th Ave. NE, Suite 208
Bellevue, WA 98004

Principal Investigator: K.M. Peery
Telephone Number: (206) 827-3304

Short Title of Work: 3D N.S. on Multiprocessor
Effective Date of Contract: 14 SEP 1989
Contract Expiration Date: 28 FEB 1989
Reporting Period: Final Technical Report
Distribution: Approved for public release; distribution unlimited.

DISCLAIMER

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AEI-TR-85290.01	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Amtec Engineering, Inc. 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army Missile Command
6c. ADDRESS (City, State, and ZIP Code) 11820 Northup Way, Suite 200 Bellevue, WA 98005	7b. ADDRESS (City, State, and ZIP Code) Redstone Arsenal, AL 35898-5244
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agcy. 8b. OFFICE SYMBOL (If applicable) ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAH01-88-C-0405
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22209-2308	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A 3D Navier-Stokes Flow Analysis For A Large-Array Multiprocessor

12. PERSONAL AUTHOR(S)
Kelton M. Peery

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 88SEP14 TO 89FEB29	89 APR 17	59

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Parallel processing; Computational Fluid Dynamics; Navier-Stokes Equations

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Computational fluid dynamics (CFD) software is an important analysis tool for engineers, particularly in the aerospace industry. CFD codes are, however, severely limited by the speed of current supercomputers -- several orders of magnitude increase in floating-point speed is necessary to effectively utilize CFD tools in engineering design. Parallel processing computers will be available in the next few years with theoretical speeds approaching that required for effective use of CFD. Much work needs to be accomplished in developing CFD algorithms to make efficient utilization of these parallel computers. In this work one explicit and two implicit algorithms for solving the 3D Navier-Stokes equations were developed and benchmarked on the Encore Multimax, a shared-memory Multiple Instruction Multiple Data (MIMD) computer. Parallelism was obtained with domain decomposition. Parallel efficiencies ranged from 50 to 95% with 2 to 9 processors on 24 x 12 x 12 and 50 x 30 x 30 meshes.

Keywords:

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Hugh C. Carson	22b. TELEPHONE (Include Area Code) (205) 876-7215
	22c. OFFICE SYMBOL AMSMI-RD-DP-TT

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Contents

1 Introduction

Computational fluid dynamics (CFD) is becoming a major element in the aerodynamic design and analysis of full aircraft and aircraft components. As computers and solution algorithms become even faster, numerical analysis will gradually replace wind-tunnel testing in most design procedures in the aerospace industry. The emphasis on wind tunnel testing will shift from parametric testing of candidate design configurations to validation of CFD numerical analyses and verification of final designs. CFD analysis can potentially generate design data much faster and at substantially less cost than by using wind-tunnel testing. CFD also offers the capability of numerically simulating flow fields that can not be (or are extremely difficult to be) achieved in wind tunnels. Since the hypervelocity high-temperature flows are very difficult to achieve in wind tunnels, CFD analysis is of critical importance in the design of hypersonic aircraft such as the National Aerospace Plane (NASP).

CFD analyses require immense computer resources. The future success of CFD analysis depends greatly upon the development of faster computers and better solution algorithms. Solutions of the 3D Navier-Stokes equations (modeling the viscous flow of compressible fluids) run for hours on present supercomputers (such as the Cray-X-MP, Cray-II, and the Cyber 205) for the analysis of flow about relatively simple aircraft components. Peterson [1] projects that computers capable of well over one teraflops (one trillion floating-point operations per second) and having at least one billion words of memory will be necessary for accurately simulating whole aircraft turbulent flow fields using current algorithms. The estimates are even more awesome when additional complexities are introduced into the equations modeling the flow field: multi-species chemical reactions, multiple phases, sub-grid scale turbulence models, and direct simulation of turbulence. The (theoretical) technological limit of the Single-Instruction-Single-Data (SISD) and the Single-Instruction-Multiple-Data (SIMD) computer architectures currently used on most supercomputers, however, is only about one gigaflop (one billion floating-point operations per second) [2]. New computer architectures must be utilized to achieve the substantial increases in computing speed required for the desired CFD applications.

The only clear path to achieving substantial increases in computing speeds is the implementation of Multiple-Instruction-Multiple-Data (MIMD) computer architectures — that is parallel computers. Numerous first generation parallel computers have been built and/or are presently available. These include the four-cpu Cray-X-MP, ILLIAC-IV, Denelcor's HEP computer, Alliant's FX/8, Intel's Hypercube, and Encore's Multimax. Unfortunately, the current algorithms have been developed primarily for SISD machines with some use of vectorization. These algorithms will not be able to utilize the full capabilities of parallel computers efficiently.

There is an urgent need to develop parallel algorithms for CFD flow analysis codes to take advantage of parallel computers, but development of parallel algorithms is not a straight forward process. One of the problems with the use of parallel computers is that their architectures differ in many respects. The number of CPU's can vary from 2 to tens of thousands and have greatly differing computing capability. The memory may be globally shared by all CPU's or each CPU may have its own dedicated memory. The method by which the CPU's communicate is also a variable. Each CPU may have a dedicated path to all of the memory, a switching network that connects the CPU's to memory, a bus that provides a common path connecting all the CPU's with the memory, or other methods as well. The above architectural aspects of parallel computers, and the fact that these architectures are changing in time, make the development of parallel algorithms difficult.

The goal of this work is to develop a flow analysis procedure for solving the three-dimensional Navier-Stokes equations. The flow analysis procedure will be capable of simulating three-dimensional viscous hypersonic flows over complex aerodynamic bodies, including the effects of finite-rate chemical reactions. Specific applications would include hypersonic vehicles like the National Aerospace Plane (NASP), SDI interceptors, as well as other conventional aircraft flow fields. The flow analysis procedure will utilize an efficient parallel algorithm for efficient computing on a large-array multiprocessor (LAmP) computer with globally shared memory.

The Navier-Stokes equations provide an accurate mathematical model of the flow of gases over most aerodynamic bodies at speeds from 0 to Mach 5 (and even higher when chemical reaction equations are included). If engineers could solve these flow equations accurately and timely (like within an hour) for partial or full aircraft configurations, the aircraft design process would be revolutionized. Engineers could rapidly evaluate candidate configurations, and explore radically different designs quickly and inexpensively. If numerical flow solutions could be performed in much less than one hour, then software could be developed to perform automated optimization of aircraft aerodynamic designs. The results would be more fuel-efficient and higher-performance aircraft that cost less to design as compared to today's aircraft.

In the present work (Phase I) several candidate parallel algorithms were developed and implemented into a prototype 3D Navier-Stokes code. The algorithms were developed on an Encore Multimax computer [3] with 10 NS32332 32-bit microprocessors, each capable of executing 2 million instructions per second (mips), yielding a total of 20 MIPS (millions of instructions per second) and approximately 3 MFLOPS (millions of floating-point operations per second) for the Linpack double precision benchmark. After incorporating the algorithms into prototype computer codes, the codes were applied to a computationally demanding flow field calculation on the Multimax. The results of this work is reported herein.

In Phases II and III the most promising parallel algorithm will be refined and optimized. The algorithm will be incorporated into a production code capable of simulating hypersonic flows over complete aircraft with complex geometry. The final result of Phases II and III will be a useful CFD flow analysis code that efficiently utilizes the parallel processing capabilty of MIMD computers with large numbers of processors and can be applied to the most computationally demanding flow field problems.

2 Objectives

2.1 Overall Objectives

The overall objective of Phases I, II, and III is to develop a commercial version of a 3D Navier-Stokes code that runs efficiently on MIMD computers with a large array of processors. When used on future MIMD computers, this code will be capable of simulating hypersonic flow over complete aircraft in a time frame that is acceptable for use in engineering design processes (approximately one hour per calculation).

2.2 Phase I Objectives

- Develop several parallel algorithms for solving the three-dimensional Navier-Stokes equations. Each will be incorporated into a prototype computer code and applied to a computationally demanding flow field calculation on a Multimax shared-memory parallel computer. The performance of the parallel algorithm will be measured and evaluated.
- Apply the above algorithms to the same flow field problem on the Dual-Multimax — a LAmP system provided by Encore Computer Corporation. The performance of the parallel algorithm will be measured and evaluated.

The results from achieving the above objectives should provide insight into the difficulties and computational advantages of several parallel algorithms for solving the 3D Navier-Stokes equations for real flow problems on a shared-memory MIMD computer.

3 Related Work

3.1 General Parallel Algorithms

As adeptly noted by Ortega and Voigt[24] there have been few truly new algorithms developed for solving partial differential equations on parallel processors. In their review they found that most parallel algorithms have been developed by restructuring the computational domain into independent portions (domain decomposition) and reordering the unknown variables to enhance the decomposition. A clear example of this approach was given by Evans[25]. Evans describes checkerboard Successive Over Relaxation (SOR) as a parallel version of classical SOR. By simply reordering a rectangular array of grid points into two sets (black points $i + j = even$ and red points $i + j = odd$) a good sequential algorithm was transformed into a good parallel algorithm. By contrast one might select simultaneous over relaxation (point Jacobi with over relaxation) as a parallel version of SOR. But as Gibbons et al.[28] states, replacing a good sequential algorithm (SOR in this case) with a poor parallel algorithm (point Jacobi) makes no sense, even though point Jacobi has high parallelism and is very easy to program on a parallel computer. The reason for this is that SOR is substantially more efficient on uniprocessors than point relaxation. The optimal parallel algorithm is one that achieves a speedup over the best sequential algorithm equal to the number of processors. Therefore, if one could efficiently parallelize the best sequential algorithm, he would have the optimal parallel algorithm. Linden, et al.[35] support this same arguement. They stress the point that parallelism by itself is not enough; parallel algorithms that are also efficient on sequential computers will be favored.

Evans [25] presented results of "group explicit" methods for creating parallelism in a relaxation algorithm for Laplace's equation. In this method the computational domain is divided into blocks of equal number and structure of cells. The equations within each block are analytically combining to eliminate dependency of the unknowns upon each other; each unknown is a function of the unknowns in the cells on the boundary of the block. In this manner the relaxation iteration can be performed on each node independently and still retain implicitness locally within each block. These methods appear to be very good for simple linear equations, but application to the non-linear Navier-Stokes equations would be difficult to program and costly in terms of storage requirements. Another interesting point that Evans makes is that there is some advantage in having a parallel algorithm that is consistent with some sequential algorithm. Theories exist for predicting the performance of sequential algorithms, but not for parallel algorithms that have no sequential counterpart (like chaotic or folded schemes).

Many real parallel computing applications were studied, catagorized, and analyzed by Fox[33]. He notes that coarse grain functional decompostion will yield only modest speedups, since most real applications do not exhibit more than a few different functions that could proceed concurrently. He concludes that over 90% of the various scientific applications he analyzed could benefit from extension to a higher level of parallelism.

Schultz, et al.[37] are developing the LCAP-1 system, a loosly coupled array of processors. Their objective is to study a computer architecture that uses existing computers connected with high-speed channels. Using large cache memories at each processor, large memories shared by groups of processors, and large global memory shared by all processors, significant overall performance was obtained on problems that could be parallelized with domain decomposition. The critical importance of having large local memories at each processor was stressed.

3.2 Parallel Algorithms for CFD

There have not been many papers written about parallel processing for computational fluid dynamics. And most of those are concerned with vector pipeline processors like the Cray 1, Cray X-MP, CDC Cyber203, Cyber205, and Star-100. Obviously, until recently there haven't been many multiple-instruction-multiple-data parallel computers on which to conduct research on parallel algorithms, and especially not on the more complex equations like the Navier-Stokes equations.

The first major parallel computer used for CFD research was the Illiac IV that was installed at NASA Ames Research Center in 1972. Lomax and Pulliam[26] report on parallelizing a 3D implicit Navier-Stokes code that used approximate factorization on the Illiac IV in 1982. Eberhardt and Baganoff[27] parallelized three CFD codes on a two-processor DEC VAX MIMD computer. Using what we have termed "variable domain decompostion" on the Approximately Factored implicit algorithm for the Navier Stokes equations, Eberhardt and Boganoff, were able to get speedups of 1.905 on the two-processor MIMD computer. They noted the importance of minimizing inter-processor communication and having large local memory.

Gropp and Smith[32] measured the performance of an explicit MacCormack method on a MIMD computer. They were able to achieve a speedup of 13 with 16 processors. They point out that to minimize interprocess communication the aspect ratio of the decomposed portions should be near 1.0, thereby minimizing the ratio of boundary cells (surface) to internal cells (volume). They report achieving speedups greater than the number of processors. This can happen when the entire domain is too large to fit

into a processor's cache, but the subdomains do fit. The processors are simply able to process at greater speeds due to a reduction in system overhead related to cache memory misses, etc.

Patel and Jordan[29] parallelized an SOR algorithm for the two-dimensional Navier-Stokes equations in vorticity/stream function form on a HEP MIMD computer. They applied the code to the driven cavity problem. They achieved a maximum speedup of 6.8. They used a diagonal wave-front ordering of the SOR sweep to achieve true point SOR.

Harding and Carling[30] parallelized the Navier-Stokes equations for a DAP (Distributed Array Processor) computer using explicit, ADI, and checkerboard SOR algorithms. They found that full 3D checkerboard was more effective than stacking similar 2D checkerboards for 3D calculations because of boundary condition effects.

Mandel[31] reports a case history of parallelizing a "real code". His real code was 40,000 lines long and used a free-Lagrange method for computing hydodynamic flow. He documents the troubles that they encountered in parallelizing the code. The first problem was that parallelizing the 3D code presented problems that were not evident in their model 1D code, mainly concerning memory usage. Pointers to large linear arrays occuring in low-level subroutines were found to complicate the parallel coding. Differentiating between local and global varibles was the most difficult aspect of parallel coding. They recommend, if possible, to design a parallel code from scratch instead of parallelizing an old sequential code. They feel this will avoid complicated coding and substantial debugging.

Hiromoto, et al.[34] point out the important concept that inner Do-loops should be reserved for pipeline vector processing, while outer loops used to parallelize. Data partitioning was used to achieve parallelism in most of the code and functional partitioning was used to compute boundary conditions while internal cells were still being operated upon.

Linden, et al.[35] developed a parallel multigrid solver for the incompressible Navier-Stokes equations at low Reynolds numbers on a zonal mesh. They had not run the code on a MIMD computer, but predicted that the parallel efficiency would increase as the number of mesh cells in the domain increased. Although the convergence times for multigrid on a sequential computer were very good and displayed a near linear relationship between run time and the number of cells, multigrid has not been as good for flows at high Reynolds numbers typically found in engineering problems.

A 2D Navier-Stokes code was multitasked on the Cray Y-MP by Fatoohi[36] using

microtasking, marcotasking, and autotasking (automatic multitasking). His results show a rapid drop off in parallel efficiency with increasing number of processsors. With 8 processors he measured a range of speedups from a low of 21% with a 64x64 mesh using macrotasking to a high of 79.5% with a 256x256 mesh also using macrotasking. Microtasking and Autotasking with 8 processors were only 40 to 50% parallel efficient for the two mesh sizes.

Catherasoo[38] has parallelized two explicit 3D Navier-Stokes codes for the AMETEK Series 2010 MIMD computer. His results show that efficiencies up to 95% can be obtained on a 32x32x32 mesh on a 64 node system. The parallel efficiency falls off when vector pipeline processors are attached to each node (although the run time decreases). An additional drop in parallel efficiency occurs when smaller meshes are used. When a 24x4x4 mesh is used on a 16 node system with vector processors the parallel efficiency is about 25%. This indicates the importance of communication overhead especially in distributed-memory MIMD computers with large number of processors.

Keyes[39] used domain decompositon to parallelize a 2D reacting flow problem. Comparing several solution methods that included relaxation schemes and iterative preconditioned conjugate gradient methods, he solved the flow field on an Encore Multimax. He obtained a speedup of about 10 with 16 processors (60% parallel efficiency).

4 Description of the NS3D Code

The Amtec NS3D code uses a mesh built up from multiple i, j, k ordered grids (zones) patched together. The patched grid gives a better discretization of a complex flow domain than can be achieved with a single i, j, k ordered grid. In this section the Navier-Stokes equations, the discretization method used to difference the differential equations, and finally the three parallel algorithms used to solve the difference equations in this work are presented and described.

4.1 Mathematical Model

The NS3D code solves the mass–averaged form of the Reynolds-averaged Navier-Stokes equations. These equations are given below in integral form [11].

$$\frac{\partial}{\partial t}\iiint_V U\, dV + \iint_S \vec{P}\cdot\vec{n}\, dS = 0 \tag{1}$$

where

$$\vec{P} = F_1\vec{i_1} + F_2\vec{i_2} + F_3\vec{i_3}$$

and

$$U = \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ E \end{pmatrix}$$

$$F_l = \begin{pmatrix} \rho u_l \\ \rho u_l u_1 + p\delta_{1l} - \tau_{l1} \\ \rho u_l u_2 + p\delta_{2l} - \tau_{l2} \\ \rho u_l u_3 + p\delta_{3l} - \tau_{l3} \\ (E+p)u_l - u_m\tau_{lm} + q_l \end{pmatrix}$$

$$\begin{aligned} E &= \rho\left[e + \frac{1}{2}u_l u_l\right] = \rho H - p \\ e &= C_v T \\ p &= \rho\,(\gamma - 1)\, e \\ \tau_{lm} &= \mu\left[\left(\frac{\partial u_l}{\partial x_m} + \frac{\partial u_m}{\partial x_l}\right) - \frac{2}{3}\delta_{lm}\frac{\partial u_n}{\partial x_n}\right] \\ q_l &= -k\frac{\partial T}{\partial x_l} \end{aligned}$$

Here the standard summation convention (sum over repeated indices) is followed and δ_{ij} is the Kronecker delta function ($\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ otherwise). The thermal condutivity, k, and the absolute viscosity, μ, are taken to be the sum of the laminar and turbulent values. A standard turbulence model is included in the NS3D code, but was not used during phase I of this contract.

4.2 Discretization and Flux Functions

Physically, Equation 1 represents a very simple idea: the time rate of change of mass, momentum, and energy within an arbitrarily chosen volume, V, is equal to the apparent flux of these quantities inward through the surface, S, surrounding the volume. The finite volume method consists of breaking the flow field up into a large number of nearly hexahedral finite volume cells, as shown in Figure 1, and applying

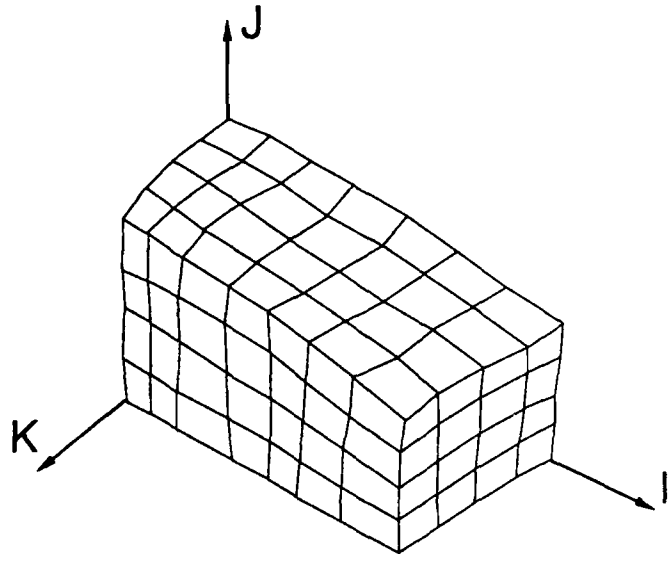


Figure 1: Finite Volume Mesh

the integral equations directly to each volume.

An individual finite volume cell, with indices i, j, and k, is shown in Figure 2. Applying the integral equations in this volume gives

$$\frac{d}{dt}(U_{i,j,k} Vol_{i,j,k}) = -\left(D_i \vec{P}\cdot\vec{S} + D_j \vec{P}\cdot\vec{S} + D_k \vec{P}\cdot\vec{S}\right)$$

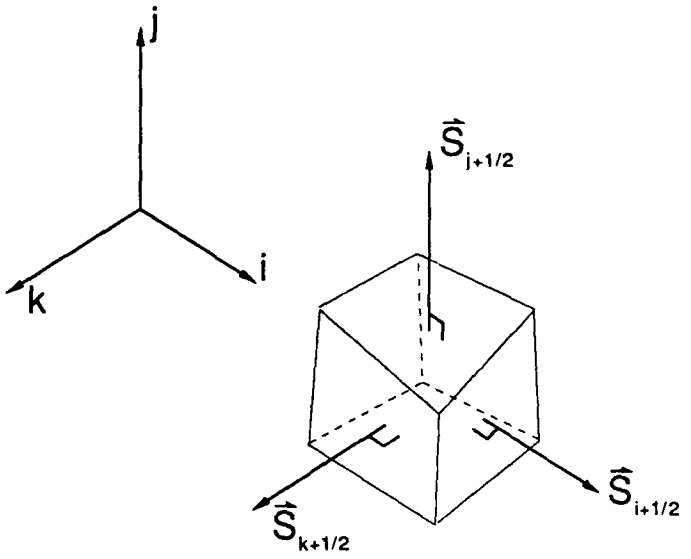


Figure 2: Finite Volume Cell

where $U_{i,j,k}$ is the mean value of U in cell i,j,k and $D_i\vec{P}\cdot\vec{S}$, for example, represents the difference of the fluxes through opposing faces of the cell.

The time derivative is approximated using backward in time differencing.

$$\frac{Vol_{i,j,k}}{\Delta t}\delta U_{i,j,k} = -(D_i\,\vec{P}\cdot\vec{S} + D_j\,\vec{P}\cdot\vec{S} + D_k\,\vec{P}\cdot\vec{S}) \tag{2}$$

where

$$\delta U_{i,j,k} = U^{n+1}_{i,j,k} - U^{n}_{i,j,k}$$

The fluxes must now be approximated in terms of the $U_{i,j,k}$. For conciseness consider only the $i+1/2$ surface.

For the approximation of the flux through a surface the inviscid and diffusion terms of the flux vector are considered separately.

$$\vec{P}\cdot\vec{S} = \vec{P}\cdot\vec{S}^{inv} + \vec{P}\cdot\vec{S}^{diff}$$

These terms are then evaluated in a manner consistent with the predominant nature of the equations in the limit as $Re \to \infty$ (hyperbolic) and $Re \to 0$ (parabolic); i.e., upwind differencing for the inviscid terms and central differencing for the diffusion (viscous stress and heat flux) terms.

The inviscid flux terms can be evaluated using any four point function (two points on each side of a cell face). In the present work we include two flux functions. The first

is based on the flux-vector-splitting method developed by Steger and Warming [12]. The second is based on Roe's approximate Reimann solver flux-difference-splitting method [13]. Second-order spatial accuracy is obtained using MUSCL-like differencing [14]. The differencing is reduced to first order flux splitting near shocks. The diffusion terms are evaluated using standard central differences [15]. The resulting flux vector is a function of the solution in four nearby cells. [1]

$$\vec{P}\cdot\vec{S}_{i+1/2} = f(U_{i-1,j,k}, U_{i,j,k}, U_{i+1,j,k}, U_{i+2,j,k}) \tag{3}$$

The exact form of the above flux function differs for Steger and Warming flux-vector-splitting and Roe's flux-difference-splitting. For details see Peery and Imlay [6] and Vinokur [16].

4.3 Solution algorithms

Equation 3 defines the spacial dependence of the surface fluxes. The temporal dependence of this flux determines the type of algorithm used: explicit or implicit. If the flux is evaluated at the known time level, n, the procedure is explicit. If the flux is evaluated at the unknown time level, $n+1$, the procedure is implicit. Evaluating the flux implicitly, and substituting into Equation 2, results in a large system of nonlinear algebraic equations which must be solved for U^{n+1} on each time step. This system is generally linearized and often simplified in other ways as well. The two implicit methods considered below differ in the degree to which this system is simplified. The explicit method determines U^{n+1} directly from U^n without solving any systems of equations.

Three solution methods were considered in this investigation: one explicit and two implicit. All three are time-marching procedures that follow the basic steps

1. Calculate residuals based upon the local variation of the solution for each cell using the explicit flux balance

$$R_{i,j,k} = -\left(D_i\,\vec{P}\cdot\vec{S} + D_j\,\vec{P}\cdot\vec{S} + D_k\,\vec{P}\cdot\vec{S}\right)^n \tag{4}$$

2. Calculate the changes in the solution, $\delta U_{i,j,k}$, from the residuals, $R_{i,j,k}$.

[1] The full form of the viscous terms including the cross derivatives cause the flux to depend on the solutions in the eight additional cells neighboring the surface ($(i+1,j+1,k),(i,j+1,k),(i+1,j,k+1)$, etc. For the $i+\frac{1}{2}$ surface). The cross derivatives are not treated implicitly, however, so this additional dependence for the flux function does not affect the implicit methods.

3. Update the solution for each cell using

$$U_{i,j,k}^{n+1} = U_{i,j,k}^{n} + \delta U_{i,j,k} \tag{5}$$

This procedure is repeated until the desired unsteady flow phenomena have been observed for unsteady problems, or a steady state is reached for steady problems.

4.3.1 Explicit Method

The explicit method is obtained by evaluating the fluxes through the surfaces of the finite-volume cells, Equation 3, using the solution at the known time level, U^n. Step 2 of the procedure on page 14 then becomes a simple rescaling of the flux residuals.

$$\delta U_{i,j,k} = \frac{\Delta t}{Vol_{i,j,k}} R_{i,j,k}$$

The explicit procedure is simple and inexpensive per time step. However, it often requires more total computer time than an implicit method.

The computer time required for a time-marching solution procedure to complete a calculation is the product of the computer time required per time step and the number of time steps required to complete the calculation. The computer time required per time step is smaller for the explicit method than for implicit methods, but the total number of time steps is often so much larger for the explicit method that it requires significantly more computer time than an implicit method. This is true because the explicit method is limited by stability to very small time steps.

To calculate the explicit time step you must first calculate the maximum allowed time step for each cell. The time step is then the smallest of the individual cell time steps. The maximum time step allowed by stability for each cell is related to the time required for an acoustic wave to cross the smallest dimension of the cell[2]. If Δy is the smallest dimension of the cell, v is the velocity in that y-direction, and c is the speed of sound the time step limitation is

$$\Delta t \leq \frac{\Delta y}{|v| + c}.$$

Therefore, if the v and c don't vary too dramatically throughout the solution domain the explicit time step is roughly proportional to the smallest cell dimension in the mesh.

[2]The maximum stable time step is also affected by the viscous terms and the other dimensions of the cell. For a complete discusion see [11].

The total number of time steps required to complete a computation is the total physical time (not CPU time) required to complete the computation divided by the time step. The physical time required to complete a computation is related to the time required for information (i.e. acoustic waves traveling at the speed of sound as well as vortical waves traveling at the velocity of the fluid) to traverse the solution domain. This latter number is relatively independent of the mesh used to solve the problem. The number of time steps for the explicit method therefore becomes large when the smallest cell dimension becomes very small in comparison to the largest dimensions of the solution domain.

Unfortunately, solutions to the Navier-Stokes equations nearly always require a mesh with extremely thin cells near the wall. This is because the viscous boundary layer is generally very thin compared to the other dimensions of the problem, and the dimension of the cells, in the direction normal to the wall, must be small compared to the thickness of the boundary layer. As a result, hundreds of thousands of time steps must often be taken to complete the calculation.

For steady state calculations, the computation time for an explicit method can often be reduced by using a different time step for each mesh cell. This allows information to propagate faster in coarser regions of the mesh and the convergence rate is often improved[3]. The dissadvantage of this approach is that the flow field does not evolve in a physically correct fashion. This often results in instabilities in the solution procedure. Of course, local time stepping *cannot* be used for unsteady solutions.

4.3.2 J-Column Implicit Method

For many problems a suitable mesh can be generated which is only refined in the j-direction (for example, normal to walls with thin boundary layers). For these problems, dramatic improvements in convergence rate may be obtained by eliminating the contribution of the j-direction fluxes to the explicit stability condition. This is done by evaluating the j-direction flux, and only the j-direction flux, implicitly and linearizing. The result is

$$\vec{P}\cdot\vec{S}^{\,n+1}_{i,j+1/2,k} \approx \vec{P}\cdot\vec{S}^{\,n}_{i,j+1/2,k} + \sum_{l=j-1}^{j+2} B^{l}_{i,j+1/2,k}\,\delta U_{i,l,k} \qquad (6)$$

where

$$B^{l}_{i,j+1/2,k} = \frac{\partial f^{n}_{j+1/2}}{\partial U_{i,l,k}}.$$

[3]The convergence rate is the change in the level of convergence divided by the number of time steps required

Substituting this into Equation 2 results in a block pentadiagonal matrix to be solved for the δU's at each i,k column of the mesh. This system is solved directly using an LU decompostion to give δU's for the entire column of cells. The off diagonals from a block upper tridiagonal matrix must be saved for the column during the decomposition process.

Memory requirements are a very important concern for the j-direction implicit method. The explicit method stores 49 real numbers for every cell in the mesh. If the Jacobians and off diagonals from the block upper tridiagonal are also saved for each cell, the storage requirement would be 224 real numbers per mesh cell. This means that a problem solved using a 50x30x30 mesh would require 10 MWords (80 MBytes) of memory as opposed to the 2.2 MWords (17.6 MBytes) that the explicit method requires. This is clearly unacceptable. The minimum memory requirement would result if the LU decomposition were performed on one column at a time and the Jacobians were calculated as they were needed for each mesh point. This would require storage for one column of off diagonals (as mentioned above) and storage for Jacobians at only one mesh cell. With this approach the increase in storage requirement for the 50x30x30 mesh would be 1625 Words (12.7 KBytes). The increase in memory is neglegible, as desired, but the resulting method is not easily vectorized or parallelized. This is because operations at adjacent cells within the column must occur sequentially. The compromise we have choosen is to perform the decomposition on all columns within a k-plane simultaneously. This requires that the Jacobians be saved for an entire j,k-row of cells and that the off diagonals be saved for an entire k-plane of cells. The resulting increase in storage requirement for the 50x30x30 mesh is 74 KWords (0.6 MBytes) of memory. This is still a small increase in required memory and the method can now be vectorized or parallelized over the i-index.

The j-direction implicit method can provide a significant reduction in computation time when compared to the explicit method. In particular, when the mesh is highly refined in the j-direction, but not the i- or k-directions, the j-direction implicit method performs very well. When the mesh is also refined in the i-direction and/or the k-direction, the i-direction and/or k-direction fluxes must also be treated implicitly.

4.3.3 J-Line Gauss Seidel Implicit Method

The j-line Gauss Seidel implicit method solves, approximately, the full system resulting from linearized implicit treatment of all fluxes. This is a block banded system of linear algebraic equations, as shown in Figure 3. The coefficient matrix for this system is very large, but is sparse and well structured. Because the system is so large

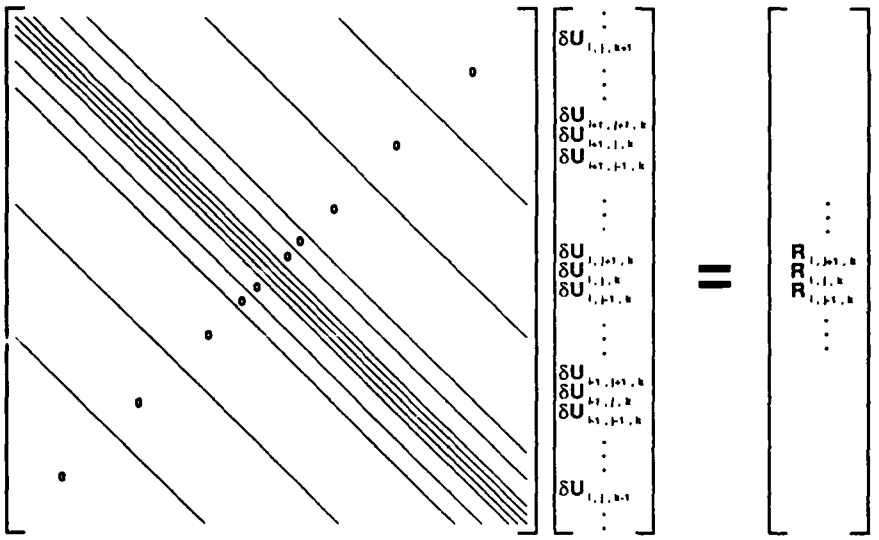


Figure 3: Block Linear System of Equations

it is impractical to solve it directly and most investigators resort to approximately factoring this matrix into simpler matrices such as block pentadiagonals. Unfortunately the error associated with the approximate factorization can severely restrict the allowable time step of these methods when the mesh is refined in two nearly perpendicular directions. In this investigation this system is solved iteratively using a modified j-line Gauss-Seidel relaxation method.

The standard j-line Gauss-Seidel relaxation method makes the system in Figure 3 solvable by multiplying the lower or upper off-diagonals by the solution at the previous iteration level and subtracting from the residuals. The implimentation of this is as follows:

1. Start with $\delta U = 0$.
2. Begin at a corner j-column of the mesh (for example, $i = 1, k = 1$).
3. Multiply the outer diagonals (any block multiplying δU's at a column other than i, k) with the appropriate δU's and subtract from the residuals on the right hand side of the equation.
4. Solve the block pentadiagonal system for $\delta U_{i,k}$.

5. Move to a neighboring column (for example, $i = 2, k = 1$).

6. Repeat steps 3 through 5 , incrementing either i or k fastest in step 5, until every column of the mesh has been done.

7. Repeat steps 2 through 6, starting with a different corner in step 2, until an accepable approximate solution to the linear system is obtained.

This relaxation procedure works well but it cannot be vectorized or parallelized. This is because the solution must be completed at an i, k column before proceeding to an adjacent column. This relaxation procedure requires five more words of memory per grid point than j-direction implicit method (a 10% increase).

The j-line Gauss Seidel relaxation was modified so that it *could* be vectorized or parallelized. The new procedure is called red, white, blue (RWB) j-line Gauss-Seidel and is similar in principle to the common multi-color relaxation procedures. The procedure is shown in Figure 4. The columns of cells are labeled so that $i = 1$ is red,

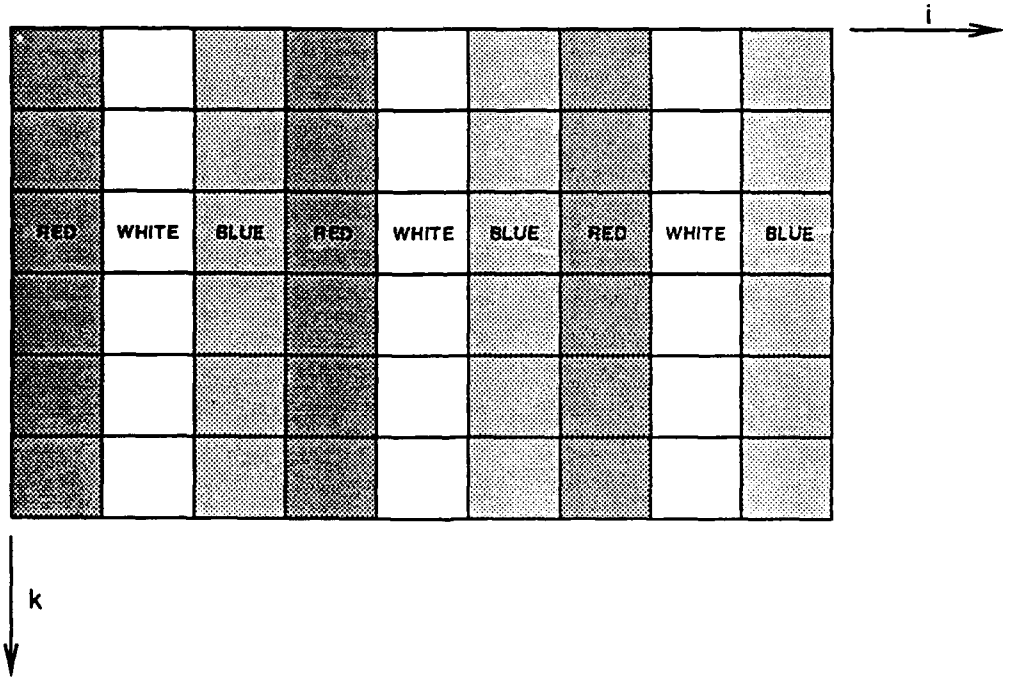


Figure 4: Red, White, Blue j-line Relaxation Procedure

$i = 2$ is white, $i = 3$ is blue, $i = 4$ is red, $i = 5$ is white, etc. Then, on each iteration,

the relaxation procedure is performed as described above for the red columns first, then the white columns, and finally the blue columns. This is done by changing the i-increment in step 5 from one to three and starting in step 2 with $i = 1$ for red, $i = 2$ for white, and $i = 3$ for blue. The advantage of the RWB relaxation over the standard relaxation is that the columns in the RWB relaxation are independent of each other during an iteration. As a result this procedure can be vectorized and/or parallelized over the i-index.

This relaxation procedure is guaranteed to converge if the linear system of Figure 3 is diagonally dominant. The system is diagonally dominant if first order upwind differenceing is used. It is not diagonally dominant with second-order upwind differencing like we use. Fortunately, diagonal dominance is a sufficient, not a necessary, condition for convergence and the relaxation generally does converge. Under-relaxation is needed though, and code must be added to detect and recover from the occasional divergent relaxation. In general, only two to four iterations per time step are needed for the time-marching procedure to be stable and converge rapidly (for steady state problems).

When the linear system of equations is solved accurately, and the time step taken is very large, this procedure approximates a Newton method for the system of nonlinear algebraic equations resulting from differencing of the time independent Navier-Stokes equations.

4.4 Description of Code Structure

The NS3D code is structured so that it is efficient, easy to modify, and easy to maintain. The subroutines are organized using a layered approach which modularizes the code according to function. This makes the code comparatively easy to debug and, therefore, reduces the time required to make a modification. The field variable data for all zones are stored in a HEAP (a one-dimensional array) in the upper level routines and are passed using pointers to the lower level routines where they are converted to multiple–dimension arrays. This results in no memory being wasted, even though the i, j, k dimensions of the different zones are unrelated. Utilities are provided for swapping zones to secondary memory and rearranging zonal data on the heap. In this manner, large problems can be accommodated by having only a small number of zones in main memory at a time.

5 Description of Encore Multimax MIMD Computers

The development of the parallel algorithms was performed on an Encore Multimax parallel computer [3]. The Multimax is a MIMD shared-memory closely-coupled computer. It may contain from 2 to 20 cpu's—each being a 32-bit microprocessor (National Semiconductor NS32332)— providing from 2 to 40 million instructions per second. The global shared memory may be up to 128MBytes. The CPU's and memory boards (which are interleaved in 8 banks to improve memory access times) are connected via a high speed bus. The bus has a maximum bandwidth of 100 MBytes per sec and includes a 64-bit data path and 32-bit address path. Each CPU has a local 64KByte high-speed memory cache to reduce CPU wait states and to reduce the bus traffic. The introduction of local cache memory creates a problem: the copy of shared data present in a CPU's local cache memory could become *stale* if another CPU writes over the shared data in the global memory. To avoid the obvious problem, Encore has incorporated a low-overhead cache coherency scheme. Each CPU has a circuit that watches the system bus. If another CPU writes to any data addresses presently in the local cache, a tag is set identifying that data address as stale. If and when the CPU trys to access stale data, it is directed to request the data from the global memory. The present work was run on Amtec's Multimax configured with 10 NS32332 processores with Weitek floating point coprocessors and 32 MBytes of random access memory.

6 Selection of Parallel Algorithms

6.1 Considerations in Selection of Parallel Algorithms for CFD

Choosing solution algorithms for parallel computers is a difficult process. The optimal choice is dependent on the type of parallel computer being used and the number of processors available. For example, an algorithm with relatively little inherent parallelism might perform well on a computer with three or four processors but very poorly on a computer with 50 processors. The choice also depends on the type, shared memory or distributed memory, of parallel computer the program will be running on. For example, on the Encore Multimax shared-memory computer we often parallelize by unrolling DO loops. With this approuch the data at a given mesh point may be operated on by one processor in one loop and another processor in the next loop. On shared memory machines with a moderate number of processors there is relatively little overhead associated with this type of operation. On a distributed memory machine, however, this would carry a larger communication overhead since data would have to be transfered from the memory of the first processor to the memory of the second before the second processor could operate on it.

There are some measures of parallel performance which are useful when evaluating parallel algorithms. The first is the speedup,

$$S_p = \frac{\text{execution time using one processor}}{\text{execution time using } p \text{ processors}},$$

which measures the speedup of a given algorithm when it is parallelized. Ortega [24] points out that one is often more interested in how much faster a given *problem* can be solved with p processors than how much faster a given *algorithm* runs with p processors. He therefore defines a modified speedup,

$$\hat{S}_p = \frac{\text{execution time using the fastest sequential algorithm on one processor}}{\text{execution time using } p \text{ processors}}.$$

This definition emphasizes the tradeoffs made to have an algorithm which can be easily parallelized. This point is of significant importance in the solution of the Navier-Stokes equations where the most easily parallelized algorithm, the explicit method, often performs very poorly compared to less easily parallelized algorithms, such as the various implicit methods. One final measure of performance is the efficiency,

$$E_p = \frac{S_p}{p}.$$

When discussing the efficiency we are generally most interested in the performance of a given *algorithm*, so we use S_p rather than $\hat{S}_p$.

Another parameter of importance is the fraction of work, α, which can be processed in parallel. Ware [41] developed a formula,

$$S_p = \frac{1}{(1-\alpha) + \frac{\alpha}{p}} \tag{7}$$

which presents an optimistic estimate of the variation in algorithm speedup with the number of processors, p, and α. Figure 5 gives the variation in S_p, given by

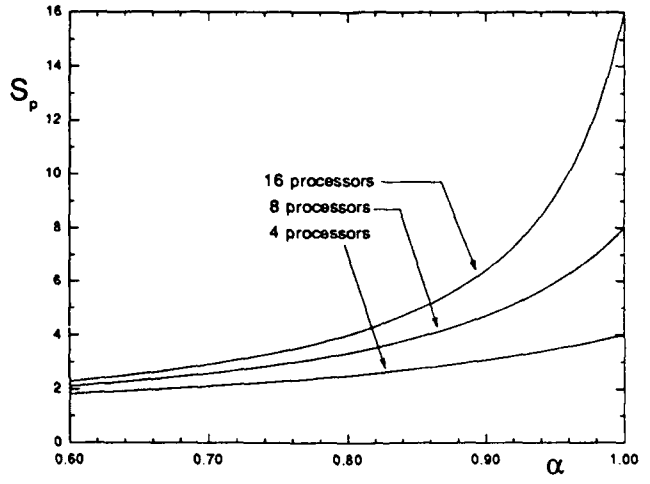


Figure 5: Speedup as a function of parallel fraction

Equation 7, with α for four, eight, and sixteen processors. It is clear from this figure that it is very important to maximize α since the speedup varies with the square of α. One weakness of Equation 7 is that it neglects the various overheads (such as communication overhead) associated with parallel computation. Another weakness is that it assumes α is not dependent on p which, as we will see in the next section, is not generally true.

There are two basic approaches for parallel algorithms: data partitioning and functional partitioning. With data partitioning, multiple identical tasks are created which operate on portions of the data. With functional partitioning, however, multiple unique tasks are created to perform different operations on the data. Data partitioning is the procedure most useful in CFD where the data at hundreds of thousands of mesh points is being manupulated in essentially the same fashion.

The majority of parallel algorithms used in CFD today are sequential algorithms which have been modified. Ortega [24] gives two techniques for parallelizing sequential algorithms which he refers to as *divide and conquer* and *reordering*. Divide and conquer refers to breaking the problem up into smaller subproblems which may be treated independently. This includes both data partitioning and functional partitioning. Reordering, on the other hand refers to "restructuring the computational domain and/or the sequence of operations in order to increase the percentage of the computation that can be done in parallel". In other words, reordering entails looking for hidden parallelism in the problem and exploiting it. One example of reordering is Patel and Jordon's [23] diagonal wave front ordering of a point SOR sweep. Some other examples of reordering are demonstrated in the next section.

6.2 Model Problem Results

The model problem is Laplace's equation ($\nabla^2\phi = 0$) within a three dimensional rectangular box. Dirichlet boundary conditions are used with ϕ set to unity on one of the six faces of the rectangle and zero on the other five faces. The problem is solved using standard second differences and point Jacobi iteration with under-relaxation. A constantly spaced mesh (unit spacing) with dimensions 30x30x50 is used.

This problem was used to study choices for parallelizing nested DO loops with DOALL — END DO statements. This problem is convenient for this study since the bulk of the work is done within the single nested DO loop shown in Figure 6. This DO

```
      do 100 k=2,kmax-1
        do 90 j=2,jmax-1
          do 80 i=2,imax-1
            a(i,j,k,np1) = (1.0-cc)*a(i,j,k,n)
     &                   + cc*(a(i+1,j  ,k  ,n)+a(i-1,j  ,k  ,n)
     &                        +a(i  ,j+1,k  ,n)+a(i  ,j-1,k  ,n)
     &                        +a(i  ,j  ,k+1,n)+a(i  ,j  ,k-1,n))/6.0
   80     continue
   90   continue
  100 continue
```

Figure 6: Model problem DO Loop before parallelization

loop is executed on every iteration along with three assignment statements. There is also some code to set the boundary conditions and initial conditions. This latter

code, which we'll call the initializing code, is only executed once at the beginning of the calculation and it's contribution to the total run time becomes less significant as the number of iterations is increased. We have, therefore, parallelized only the nested DO loop shown in Figureẽffig:ModelCode0. For the calculation of efficiency, we eliminate the contribution of the initializing code by considering the *difference* of times for a 50 iteration run and a 100 iteration run.

Four approaches to parallelizing this loop were considered. They span the range from very large data partitions to very small data partitions. The expectation at the beginning of this study was that the cases with large data partitions would yield the highest efficiencies on our Encore Multimax computer with six processors. This is because the inherent overhead associated with scheduling parallel tasks is less significant when there are fewer tasks (larger data partitions). However, it was also recognized that fewer tasks could result in significant processor idle time if the number of processors does not divide evenly into the number of tasks. Our purpose for studying a simplified model problem was to quantify the above effects so that intellegent decisions could be made when parallelizing the Navier-Stokes code.

The first approach to parallelizing this nested DO loop, (case 1), is to simply replace the outermost DO loop, the loop over the k-index, with a DOALL — END DO loop, (See Figure 7). This effectively partitions the data into $kmax - 2$ planes of data, each of which may be assigned it's own processor. These are large partitions relative to the number of processors. The advantage of this approach, as stated earlier, is that each task contains a relatively large amount of data to work on so that any overhead involved in scheduling parallel tasks should be insignificant.

Contrast case 1 with case 2, (see Figure 7), where the parallelization occurs over the innermost loop. In case 2 the data partitions, and the amount of work contained within each task, is small so the overhead involved in scheduling parallel tasks will be more significant than in case 1. For the 30x30x50 mesh considered here, case 1 yields 48 data partitions and case 2 yields 37,632 data partitions.

One of the major problems with cases 1 and 2 is that they cannot effectively use large numbers of processors. For the 30x30x50 mesh, they cannot use more than 48 processors. Furthermore, the parallel performance shows significant degradation when the number of processors used does *not* divide evenly into the range of the DOALL loop, 48. For example, if 23 processors are requested, 21 of them will sit idle for 1/3 of the time. This means that the maximum efficiency[4] for 23 processors is 69.57%. A plot of the maximum efficiency versus the number of processors is shown in Figure 8 for cases 1 and 2. This processor idle time is not very significant for

[4]Maximum efficiency meaning, in this case, the efficiency obtained if there is no overhead associated with parallel execution.

```
------------------------------------- OUTER LOOP PARALLELIZED - CASE 1
        doall (k=2:kmax-1)
          do 90 j=2,jmax-1
            do 80 i=2,imax-1
              a(i,j,k,np1) = (1.0-cc)*a(i,j,k,n)
     &                     + cc*(a(i+1,j  ,k  ,n)+a(i-1,j  ,k  ,n)
     &                          +a(i  ,j+1,k  ,n)+a(i  ,j-1,k  ,n)
     &                          +a(i  ,j  ,k+1,n)+a(i  ,j  ,k-1,n))/6.0
   80       continue
   90     continue
        end doall

------------------------------------- INNER LOOP PARALLELIZED - CASE 2
        do 100 i=2,imax-1
          do 90 j=2,jmax-1
            doall (k=2:kmax-1)
              a(i,j,k,np1) = (1.0-cc)*a(i,j,k,n)
     &                     + cc*(a(i+1,j  ,k  ,n)+a(i-1,j  ,k  ,n)
     &                          +a(i  ,j+1,k  ,n)+a(i  ,j-1,k  ,n)
     &                          +a(i  ,j  ,k+1,n)+a(i  ,j  ,k-1,n))/6.0
            end doall
   90     continue
  100   continue
```

Figure 7: Coding for test cases 1 and 2

systems with fewer than 10 processors but becomes more significant as the number of processors increases.

The idle time described above can be mostly avoided if smaller data partitions are used. This is done in cases 3 and 4, (See Figure 9), Case 3 replaces all three DO loops by a single DOALL loop over a parameter, m. The indices i, j, and k are then calculated explicitly in terms of m. Case 3 virtually eliminates the idle processor problem described above, at least until thousands of processors are considered, but it adds expense for the calculation of the indices and it uses very small data partitions. Case 4 is a compromise between cases 1 and 3. It replaces the j and k DO loops with a DOALL over m and retains the innermost DO loop over i. Indices j and k must still be computed from m but the calculations are simpler than for case 3 and they are performed fewer times. The data partition is also larger, with associated benefits. The only disadvantage of case 4 over case 3 is that the idle processor problem occurs with fewer processors. With case 4 the maximum efficiency first drops below 90% with 166 processors whereas it takes 4646 processors for case 3 to drop below 90%.

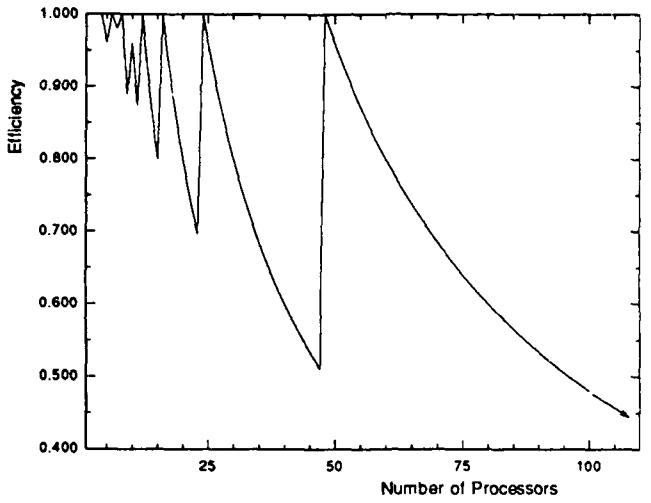


Figure 8: Maximum efficiency variation with number of processors - model problem cases 1 and 2

However, these numbers are both large in comparision to the 9 processors required for the maximum efficiency of cases 1 and 2 to first drop below 90%. Plots of maximum efficiency versus number of processors are given in Figure 10 for case 3 and Figure 11 for case 4.

The parallel efficiencies and run times for all four cases are shown in Figure 12. Case 1 retained a very high efficiency when the number of processors divided evenly into 48 with predicted lower efficiencies otherwise. Case 2 yielded somewhat lower efficiencies than case 1, 2.5 percent less efficient with 6 processors, but had the same trends. Case 3 eliminated the "idle processor" dip in the efficiency with 5 processors and had efficiencies comparable to case 2 elsewhere. This was expected since the two cases have similarly sized data partitions. Case 4 also eliminates the "idle processor" dips and performs better than case 3. This was also expected since case 4 has larger data partitions than case 3.

One surprising thing about the results is just how efficient cases 2 and 3 really are. The data partition for these cases is very small. In case 2 each task contains just one statement, with seven floating point addition/subtractions, two floating point multiplies, and one floating point divide. Each task in case 3 adds to this eleven integer additions/subtractions, three integer multiplies, and two integer divides. This is very few operations compared to the average nested loop in the Navier-Stokes code. There are many places in the implicit part of the Navier-Stokes code where

```
---------------------------------- SINGLE LOOP PARALLELIZATION - CASE 3
        doall (m=1:mtot)
          k = (m-1)/ijtot + 2
          j = (m-(k-2)*ijtot-1)/itot + 2
          i = 1 + m - (k-2)*ijtot - (j-2)*itot
          a(i,j,k,np1) = (1.0-cc)*a(i,j,k,n)
     &                 + cc*(a(i+1,j  ,k  ,n)+a(i-1,j  ,k  ,n)
     &                      +a(i  ,j+1,k  ,n)+a(i  ,j-1,k  ,n)
     &                      +a(i  ,j  ,k+1,n)+a(i  ,j  ,k-1,n))/6.0
        end doall

------------------------------------ TWO LOOP PARALLELIZATION - CASE 4
        doall (m=1:mtot)
          k = (m-1)/jtot + 2
          j = 1 + m - (k-2)*jtot
          do 80 i=2,imax-1
            a(i,j,k,np1) = (1.0-cc)*a(i,j,k,n)
     &                   + cc*(a(i+1,j  ,k  ,n)+a(i-1,j  ,k  ,n)
     &                        +a(i  ,j+1,k  ,n)+a(i  ,j-1,k  ,n)
     &                        +a(i  ,j  ,k+1,n)+a(i  ,j  ,k-1,n))/6.0
   80     continue
        end doall
```

Figure 9: Coding for test cases 3 and 4

data dependences force us to use small data partitions to parallelize DO loops. These results indicate that little efficiency is lost on the Encore Multimax by doing this.

The run time results for 100 iterations, (See Figure 12) show significant differences in the single stream (1 processor) run times. Case 4 is fastest at 316 seconds, followed by case 1 at 327 seconds, case 2 at 334 seconds, and finally case 3 at 385 seconds. The slow speed of case 3 is almost certainly due to the computation of the i, j, and k indices. The fact that case 4 is faster than case 1, single stream, is unexpected since it has five more integer additions/subtractions, one more integer multiply, and one more integer divide than case 1.

It is clear that the approach taken in case 4 is the best of those studied. It is the fastest of the four cases, has one of the highest processor efficiencies, and is likely to perform well with increasing processors. This is the approach taken in the Navier-Stokes code whenever possible. It is comforting to know, however, that those regions of the program which must be coded like case 2 still perform reasonably well on the Encore Multimax with up to ten processors.

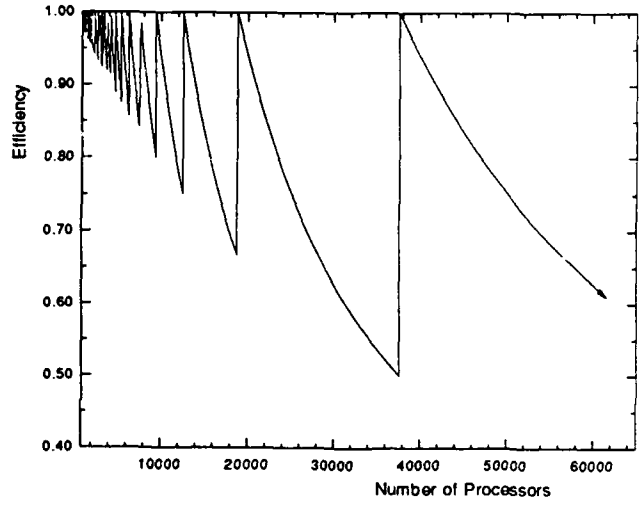


Figure 10: Maximum efficiency variation with number of processors - model problem case 3

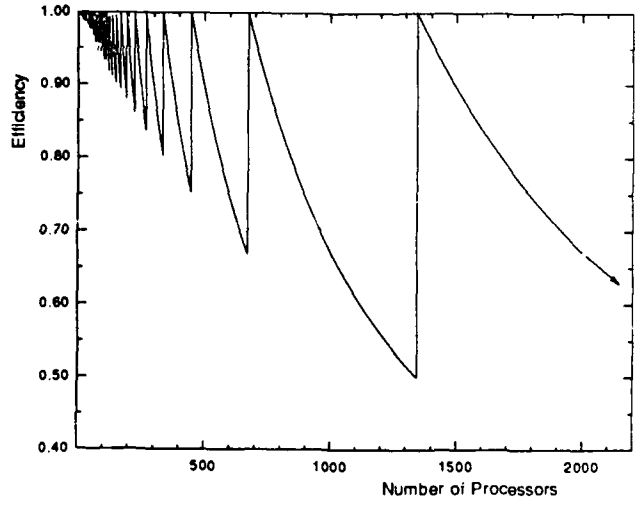


Figure 11: Maximum efficiency variation with number of processors - model problem case 4

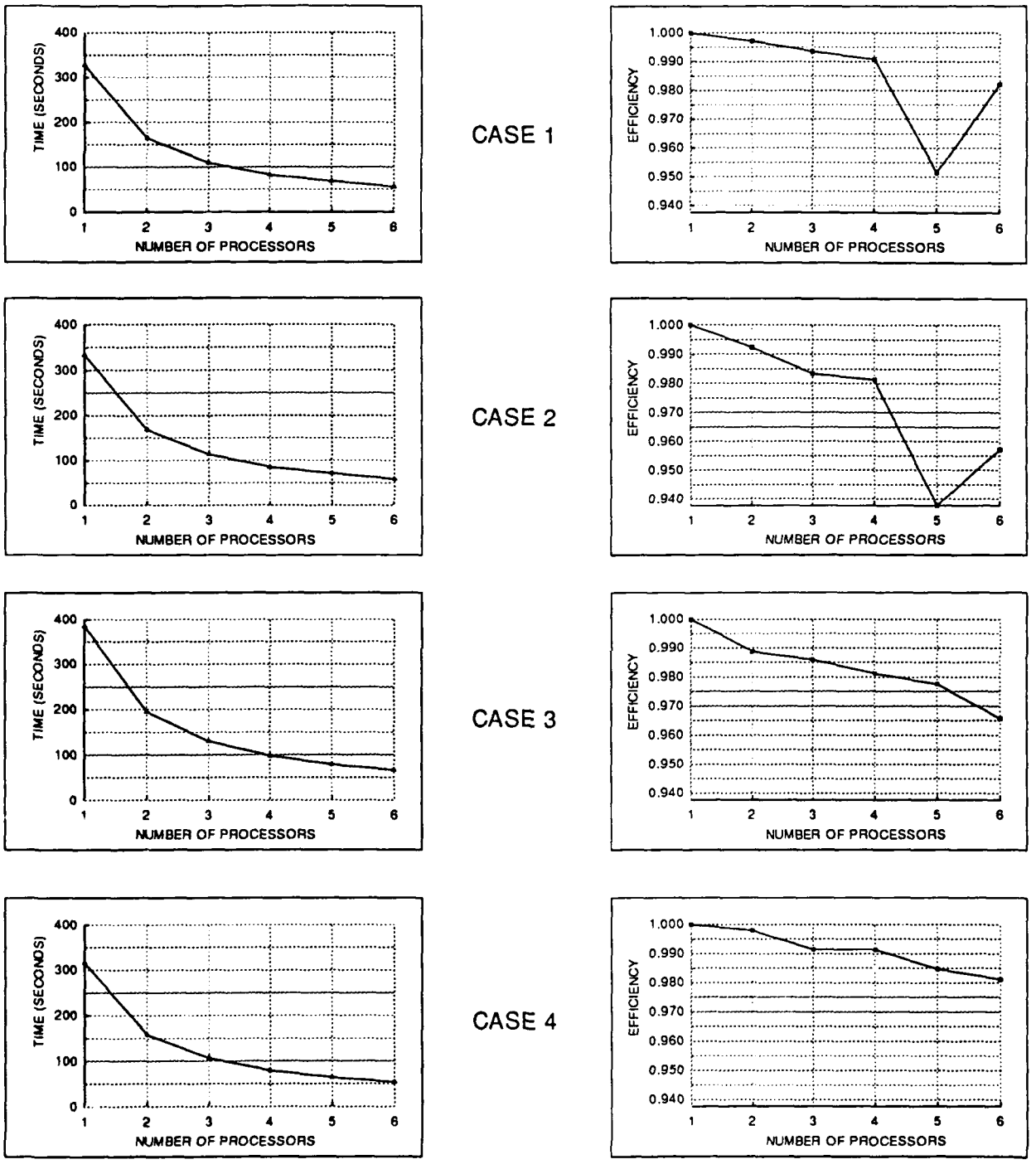


Figure 12: Comparison of measured efficiencies and run times for model problem cases 1 through 4

7 Parallelizing the NS3D Program

The NS3D program was originally written using the standard FORTRAN 77 language. The computer used for most of the development of NS3D was a Multimax 310 minicomputer made by Encore.

7.1 The EPF Compiler

Programming on the Encore Multimax 310 was done with the Encore Parallel FORTRAN compiler (EPF). The EPF compiler has the following characteristics:

- Supports automatic or user-specified parallelism.
- The number of processes used to execute a given program are specified by the user prior to code execution and can be any number less than or equal to the number of CPU's available.
- Parallelism can be confined to localized sections of code or can be global. This flexibility allows the parallel coding to be tailored to meet the design of the original program. Programs with a high degree of parallelism can make use of the global constructs whereas programs that contain isolated sections of parallelism need only parallelize those sections.
- Data can be shared or private. Shared data can be accessed and modified by all processes within a parallel region. Private data is allocated for each process and can only be accessed and modified by the process to which it is assigned.
- Processes are assigned using the master/slave model. At program startup a master (or root) process is spawned. When a parallel region of code is reached the slave processes are released along with the master process to execute all code within the parallel region.
- Process partitioning can be done dynamically (at runtime) or statically. Dynamic partitioning occurs when the work assigned to each process is done at runtime and may vary from one run to the next. Static partitioning is when the work assigned to each process is determined prior to code execution.

7.1.1 EPF Program Model

A program can be divided into regions in which processes will be spawned and run in parallel. The entire program can be run in parallel or any number of code segments within the program can be designated to run in parallel. For most programs there is a mixture of code segments where some are easily parallelized and others are best run single stream. If this is the case then the overhead associated with process synchronization can be avoided for the non-parallelizable regions by parallelizing only those regions best suited for multiple processes.

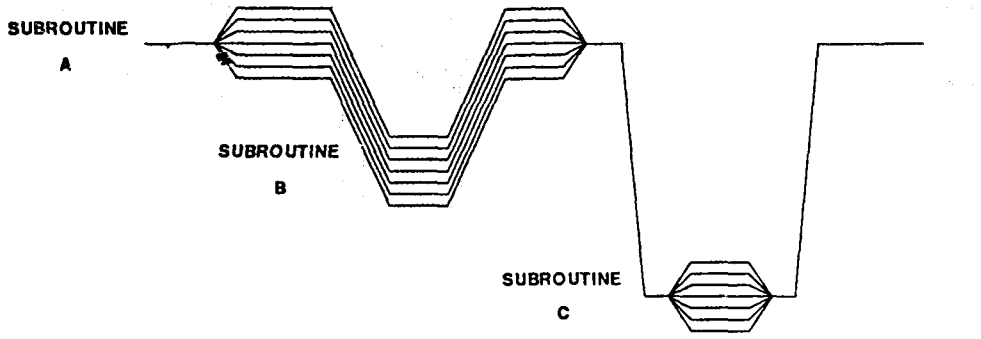


Figure 13: Typical program flow showing localized parallel regions

Figure 13 shows the process flow for three subroutines written using EPF FORTRAN. A single line represents sections of code where only the master process is executing. Multiple lines in parallel represent sections of code where all slave processes and the master process are executing. In Figure 13, Subroutine A starts out in single stream (only the master process is active). A parallel region is reached and the slave processes are spawned. Within the parallel region is a call to subroutine B and so all processes follow the call to B and then return. The end of the parallel region in A is reached and code execution becomes single stream. The call to subroutine C is made while A is in single stream and subroutine C itself contains a parallel region.

Within a parallel region each process will execute each statement unless directed to do otherwise. Various constructs are available to partition the processes. For example the DOALL - END DOALL constuct will partition the iterations of a FORTRAN DO loop among all available processes.

7.1.2 Memory allocation

Figure 14 shows how memory is allocated for multiple processes using the EPF compiler. The default is for all variables outside of a parallel region to be shared by

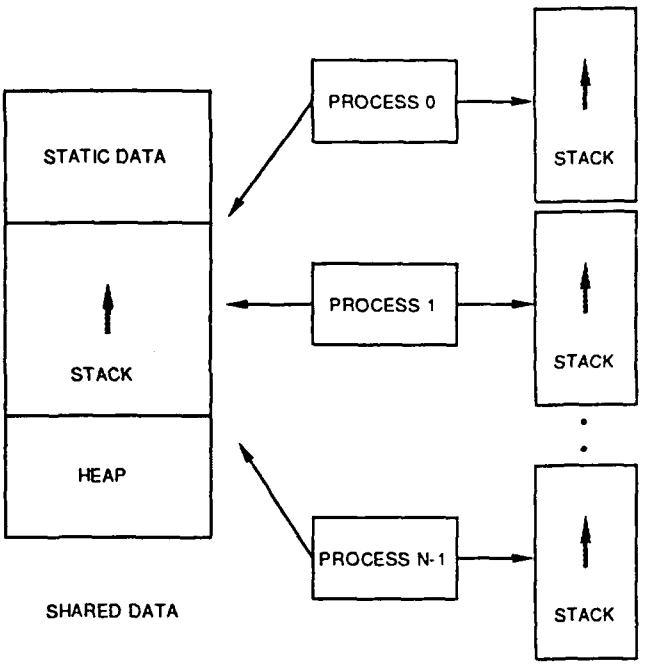


Figure 14: Memory map for multiple processes

all processes. The accessibility of all shared variables are governed by the standard FORTRAN scoping rules. A copy of all variables declared within a parallel region or variables local to a subroutine called from within a parallel region are private to each process. A variable declared as STATIC or SHARED can be accessed or modified by all processes provided that it obeys the FORTRAN scoping rules. When a parallel region is encountered the private variables are allocated onto a local stack for each process. Variables within functions called from within a parallel region are also placed on this stack and are, therefore, private to each process. The local stack used by the EPF compiler for parallel processes is limited to 256k bytes.

7.1.3 EPF compiler directives

The Major parallel constructs in EPF FORTRAN are:

- **PARALLEL**
  END PARALLEL
  The PARALLEL - END PARALLEL environment defines a section of code for which multiple processes will be made available. When the PARALLEL statement is encountered the master process spawns n-1 subprocesses, where n is user selectable prior to the execution of the program. All members of the "process set" (the master process and all spawned subprocesses) will execute all statements of the parallel block unless directed to do otherwise.

- **PRIVATE [var-list]**
  SHARED [var-list]
  PRIVATE and SHARED are used to declare how a set of variables are to be addressed. A variable that is PRIVATE is placed in the processes local stack and can be accessed and modified only by the process to which it is assigned. SHARED data is placed in the global data area and can be accessed and modified by all processes.

- **DOALL (istart: iend [:iskip])**
  END DOALL
  The DOALL - END DOALL environment defines a section of code that is analogous to the FORTRAN DO loop construct. Depending on the complexity of the loop processes may be pre-assigned to a set of loop indices or processes are assigned to loop indices in the order that the processes become available.

- **CRITICAL SECTION**
  END CRITICAL SECTION
  A CRITICAL SECTION defines a region inside of which only one process is allowed at a time.

- **BARRIER**
  A BARRIER statement causes the processes to wait until all have arrived before proceeding.

- **BARRRIER BEGIN**
  END BARRIER
  The BARRIER BEGIN - END BARRIER construct defines a region through which only one process is allowed. All other processes will skip around this region and wait at the end until the one process finishes.

- **EVENT [var-list]**
  This specification statement is used to declare a variable to be an event type.
- **LOCK [var-list]**
  This specification statement is used to declare a variable to be a lock type.
- **SEND SYNC event-name**
  WAIT SYNC event-name
  The SEND SYNC and WAIT SYNC operators are used to force processes to wait or allow them to proceed.
- **WAIT LOCK lock-name**
  SEND LOCK lock-name
  The SEND LOCK and WAIT LOCK operators are used to operate on semaphores to allow a process exclusive access to a section of code.

The built-in functions available with EPF are:

- **CONDLOCK**
  This function takes as an arguement a LOCK variable and returns .True. if the lock was successfully taken and .False. otherwise.
- **NTASKS**
  This function returns the number of processes in the current process set.
- **TASKID**
  This function returns the process-ID of the calling process.

7.2 Approach

This section descripes the approach taken to incorporate code into the existing version of the NS3D program to obtain the maximum amount of parallelization using the EPF compiler.

7.2.1 Code Evaluation

Using tools such as the profiler available on the Multimax 310, subroutines that are the most expensive in terms of CPU usage were identified. Figure 15 shows an example output from one of the profiler runs. The profiler output was generated

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
35.2	829.84	829.84	20	41492.0	_cispluj_
21.0	495.18	1325.02	840	589.50	_xssrbs_
15.0	353.92	1678.94			_cijcroe_
3.9	91.94	1770.88			_ciflxrj_
3.8	89.24	1860.12	860	103.77	_cijacj_
3.7	86.74	1946.86			_ciflxri_
3.3	78.40	2025.26			_cijcvsj_
2.8	65.86	2091.12			_ciupdq_
2.5	58.64	2149.76	20	2932.0	_cispbsj_
2.0	46.14	2195.90			_cilts_
1.3	30.82	2226.72			_pow
1.2	29.46	2256.18			_cibcfsw_
0.8	19.72	2275.90			_citurb_
0.6	13.44	2289.34	126	106.67	_cibccp_
0.4	9.66	2299.00	21	460.0	_cibcdu_
0.4	9.64	2308.64			_cimetr_
0.3	8.20	2316.84			_ciupdu_
0.2	5.70	2322.54			_cisrc_
0.1	2.40	2324.94			_ciconv_
0.1	1.74	2326.68			_cibcnsw_
0.1	1.66	2328.34			_fwrite
0.1	1.54	2329.88			_t_getc
0.1	1.50	2331.38			_s_cmp
0.1	1.44	2332.82			_cibcspi_
0.1	1.42	2334.24	21	67.6	_cibczgo_
0.1	1.32	2335.56			_sysadmin
0.1	1.22	2336.78			_ciinuc_
0.1	1.20	2337.98			_rd_int

Figure 15: Example profiler output

by running the NS3D code using a single stream (i.e. only one CPU). It is easy to see that there are three main subroutines and 10 or 12 others which performed the bulk of the work. Since this profiler output represents the work done for just one particular run of the NS3D code, other profiler runs were made with different options active. These other runs then highlighted a different set of subroutines which would also require parallelization. In most cases the subroutines which topped the profiler lists were already suspected as being dominant based on a working knowledge of the code.

7.2.2 Data Partitioning

On a global scale, the data within the NS3D program is configured in three dimensional blocks, or domains. To perform the various operations on these domains, they are divided up in a number of different ways. Figure 16 shows the original three dimensional domain along with four common ways in which the domains can be de-

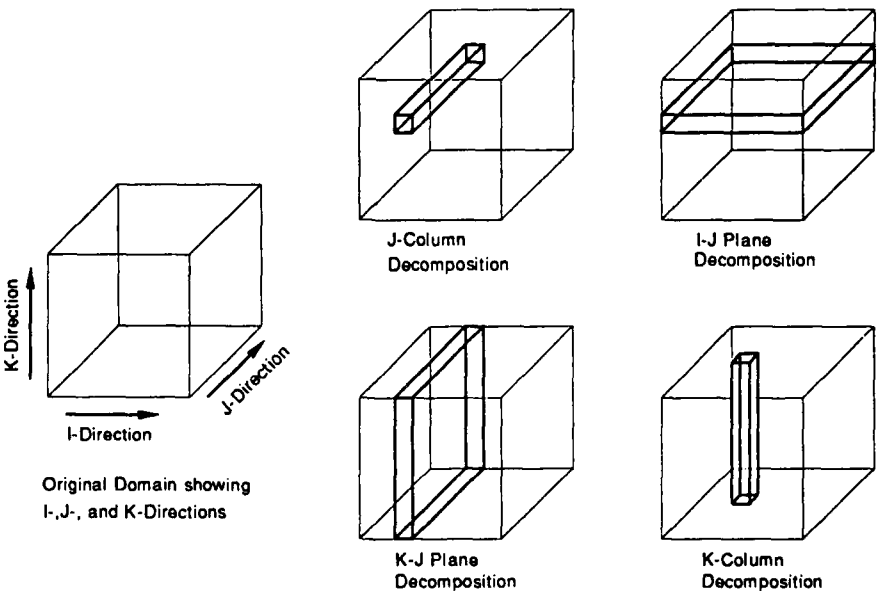


Figure 16: Domain decomposition within the NS3D program.

composed. One form of data decomposition is to hold two indices constant and vary the third. This in effect operates on a single column at a time. A second form of data decomposition is to hold one index constant and vary the other two. This in effect

Algorithm	Decomposition is governed by: Most Dominant			Least Dominant
Explicit	Maintain Vectorization	Modularity	Memory Resources	Algorithm
Implicit	Memory Resources	Maintain Vectorization	Algorithm	Modularity

Figure 17: Criterion used for domain decomposition.

operates on an entire plane at a time.

The NS3D program decomposes the three dimensional domain in a number of different ways. Figure 17 shows what factors influence how the domain is decomposed for both the implicit and the explicit algorithms.

Since the sizes of cases run using the NS3D program are usually very large, much consideration is given to the amount of memory resources available when determining how the domain is decomposed. With the implicit method a large number of temporary variables must be used at each mesh point within a subdomain. If large subdomains were used, the program would use too much memory and would not fit within the RAM of most machines. This is less important for the explicit algorithm because it uses fewer temporary variables. Another important factor in determining the domain decomposition was the need to retain loop vectorization to take advantage of machines such as the Cray. As will be demonstrated later, sometimes during the re-coding of NS3D, it was possible to maintain the vectorization for the Cray implimentation while at the same time adjust the order of a DO loop to increase the granularity for the parallel implimentation. In the explicit algorithm, since it does not matter in what order each cell of the domain is solved for a given time step, the algorithm plays very little role in the domain decomposition. The implicit algorithms however do influence the domain decomposition because of an interdependancy from one row of cells to the next.

One other major factor that must be kept in mind when parallelizing a program is the load balancing. Depending on the target machine and the dimensions of the problem to be solved, sometimes a larger granularity is better to minimize the synchronization overhead and sometimes a smaller granularity is better because it may improve the load balancing.

7.2.3 Analyze Code Flow

A hierarchical graph of the code flow was made to help identify which subroutines should and could be parallelized and in what manner. Figure 18 shows such a graph of the most costly subroutines in the NS3D code. On the graph each subroutine is marked as to what portions of its code could be parallelized and what index (i,j,k or other) would be unrolled. Also, calls made to subroutines from parallelized regions within subroutines were checked to make sure that they executed code that would not conflict with the the parallel context of the call (e.g. if a call is made within a parallel loop over the I index, care must be taken to make sure that the called subroutine doesn't rely on a sequential ordering of the I index and also that it doesn't alter any global variables that the calling subroutine depends on to remain constant).

7.3 Coding

Each subroutine marked to be parallelized was re-coded. All of the FORTRAN statements added for parallelization are initially commented out but are marked so that a preprocessor can activate them later. Also existing FORTRAN code that conflicts with the parallel coding is left intact but is marked so that the preprocessor will comment them out. In this way the original code can still compile and run as before with no alterations.

As a tool, the EPF compiler was sometimes used to automatically generate the parallelized code. By using the -F flag on the compile command a listing can be generated which shows how the EPF compiler would insert the parallelizing statements. For the most part however, the EPF compiler does a very conservative, safe job in parallelizing, and is not as efficient as parallelizing by hand. Using the automatic feature for parallelizing helps with parallelizing by hand in that it identifies for you all of the variables that need to be declared as PRIVATE for the parallel regions.

Figure 19 shows a small section of code before and after re-coding for parallelization. The **!+P** comments tell the preprocessor that this is a statement that is to be activated (i.e. remove the C in column 1). The **!-P** comment tells the preprocessor to insert a C in column 1 and comment out the line.

In the example in Figure 19, the 100 CONTINUE statement was marked with **!-P** to comment it out later although this is unnecessary. The statement: DO 100 I = IS,IE however must be marked with **!-P** so the preprocessor will comment it out later.

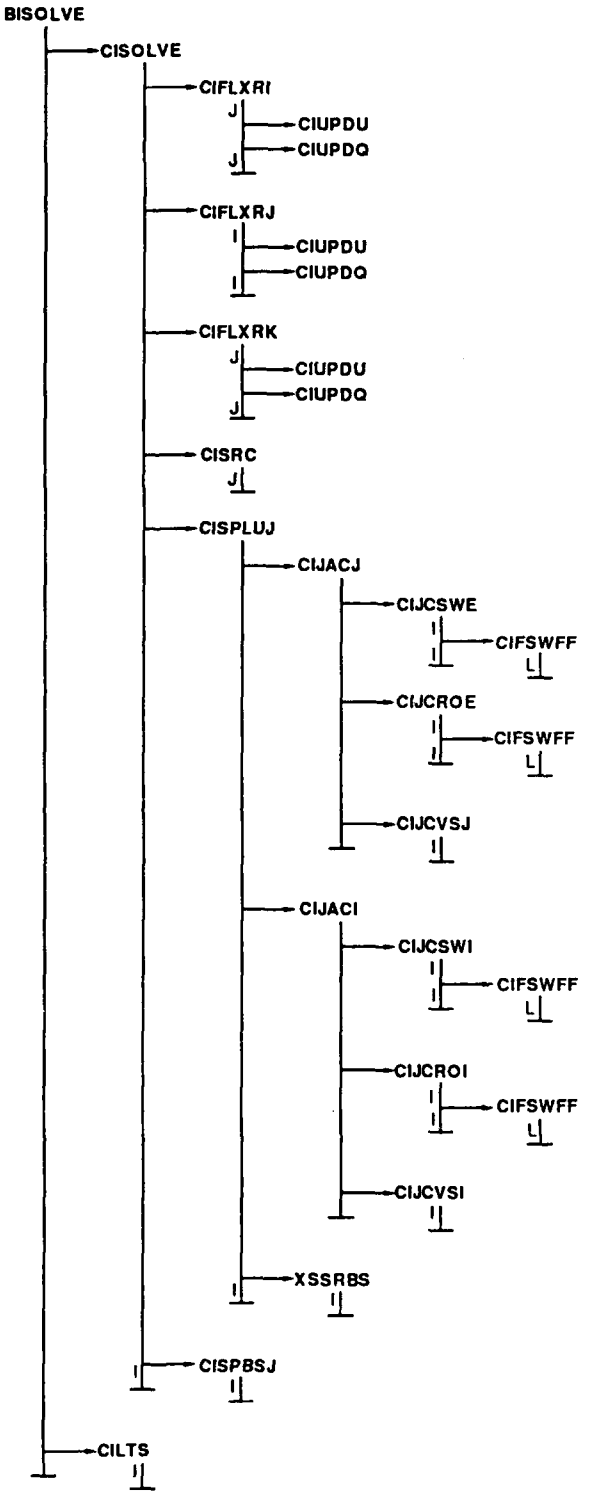


Figure 18: Hierarchical Graph of Costly Subroutines.

Original coding:

```
      .
      .
            DO 100 I = IS,IE
               X(I) = X(I)*B
        100 CONTINUE
      .
      .
```

Parallelized Coding:

```
      .
      .
            DO 100 I = IS,IE                                  !-P
      C     DOALL (I = IS:IE)                                 !+P
              X(I) = X(I)*B
      C     END DOALL                                         !+P
        100 CONTINUE                                          !-P
      .
      .
```

Figure 19: Example Code before and after adding parallel constucts.

The DOALL - END DOALL construct was used most often when parallelizing the NS3D code. Originally, the NS3D code was written in a way to best take advantage of the vectorizing capabilities on machines such as the Cray. Often the inner loop of a nested DO loop was written to be as long as possible to increase the vector length. Where possible these loops were re-ordered by moving the index to be parallelized as close to the most outer loop as possible. This in effect increases the local granularity for the DO loop. Figure 20 shows an example of a DO loop before and after index re-ordering.

The BARRIER BEGIN - END BARRIER construct was also used to guarantee that one process only would execute certain sections of code and the CRITICAL SECTION - END CRITICAL SECTION construct was used to allow only a one process at a time to execute a given section of code.

7.4 Debugging

The EPF compiler does some checking to verify that all local variables to a parallel region have been declared as being PRIVATE or SHARED. For the first attempt at compiling a new subroutine after recoding for parallelization the EPF compiler would typically find a few variables that were missed. Sometimes this checking also revealed mistakes in the original coding.

After parallelizing each subroutine a suite of test cases are run to verify that output of the parallelized version matches that of the original code (which was compiled with the f77 compiler). When the results are not the same then one or more of the following steps are taken:

- The new subroutine is recompiled with the debugger flag turned on. The program is re-run with various break points set and PRIVATE values and loop indices are checked for each process.
- Sections of the new subroutine are "spared out" to try and isolate the problem. This is done by using the BARRIER BEGIN - END BARRIER construct.
- The program is searched for variables that are not initialized properly. Fortunately/Unfortunately the EPF compiler does not initialize the local variables for each subroutine. Since this is not good programming practice (and usually not done intentionally) these mistakes are good to find. If the number of processes is reduced to one and the problem still persists then uninitialized values rather than parallel coding mistakes are most likely to blame.

Original coding:

```
        DO 520 NROW=1,NEU
          DO 520 NCOL=1,NEU
            NBCOL = NCOL + 1
            DO 520 NMULT=1,NEU
              DO 520 I=IBEGTRU,IEND,ISTRID
                BBB(I,NROW,NBCOL) = BBB(I,NROW,NBCOL)
     &             - C(I,NROW,NMULT,JTOP)*G2(I,JP,NMULT,NCOL)
  520   CONTINUE
```

Parallelized Coding:

```
C       DOALL(I=IBEGTRU:IEND:ISTRID)                                  !+P
        DO 520 NROW=1,NEU
          DO 520 NCOL=1,NEU
            NBCOL = NCOL + 1
            DO 520 NMULT=1,NEU
              DO 520 I=IBEGTRU,IEND,ISTRID                            !-P
                BBB(I,NROW,NBCOL) = BBB(I,NROW,NBCOL)
     &             - C(I,NROW,NMULT,JTOP)*G2(I,JP,NMULT,NCOL)
  520   CONTINUE
C       END DOALL                                                     !+P
```

Figure 20: Example of index re-ordering to increase local granularity.

8 Results of a Real Parallel CFD Application

8.1 Skewed Shock Wave/Laminar Boundary–Layer Interaction

In order to demonstrate the parallel NS3D code a complex 3D flow field has been calculated. The flow involves the interaction of a laminar flat plate boundary layer with the skewed shock generated by a wedge as shown in Figure 21. This case is of considerable interest in view of the current emphasis on hypersonic vehicles. A flow

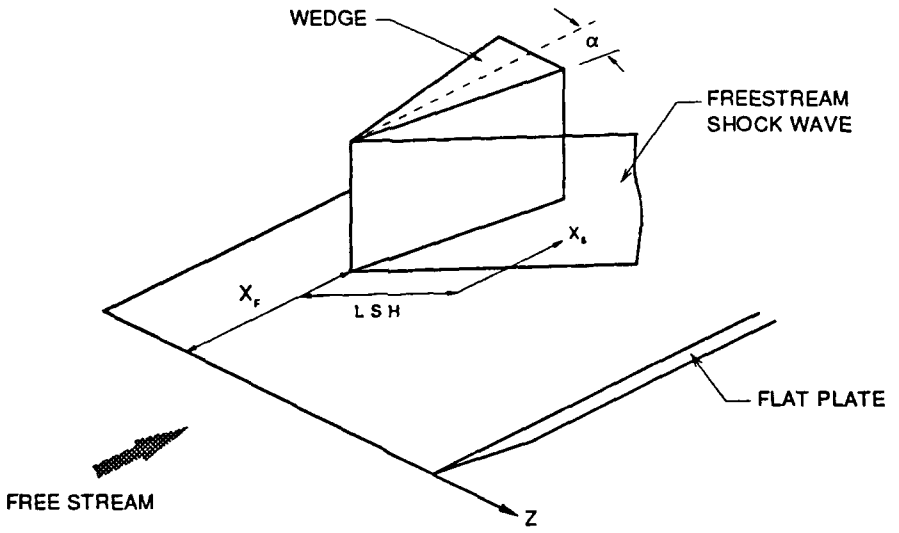


Figure 21: Interaction of laminar flat plate boundary layer with skewed shock

with a laminar boundary layer is particularly useful for validating a Navier-Stokes code since it removes the uncertainty caused by turbulence models.

The skewed shock wave/laminar boundary-layer interaction flow field was investigated experimentally by Degrez and Ginoux[40]. The freestream flow had a Mach number of 2.25 and a Reynolds number of 2.4 x 10^6 m^{-1}. The freestream temperature and pressure were approximately 153 °K and 0.0195 atm. The boundary layer was allowed to develop on the adiabatic plate for 6 cm upstream of the apex of the 8 degree wedge. This produced a boundary-layer thickness of $\sim$ 1 mm. In the interaction region the boundary layer separates along a line that runs along the shock.

The distance between the separation line and the shock is called the upstream influence, which increases with distance from the wedge apex. The data indicates that the laminar interaction has a great similarity with turbulent interactions. The separated interaction region forms a trapped vortex which carries the low momentum fluid downstream along the shock.

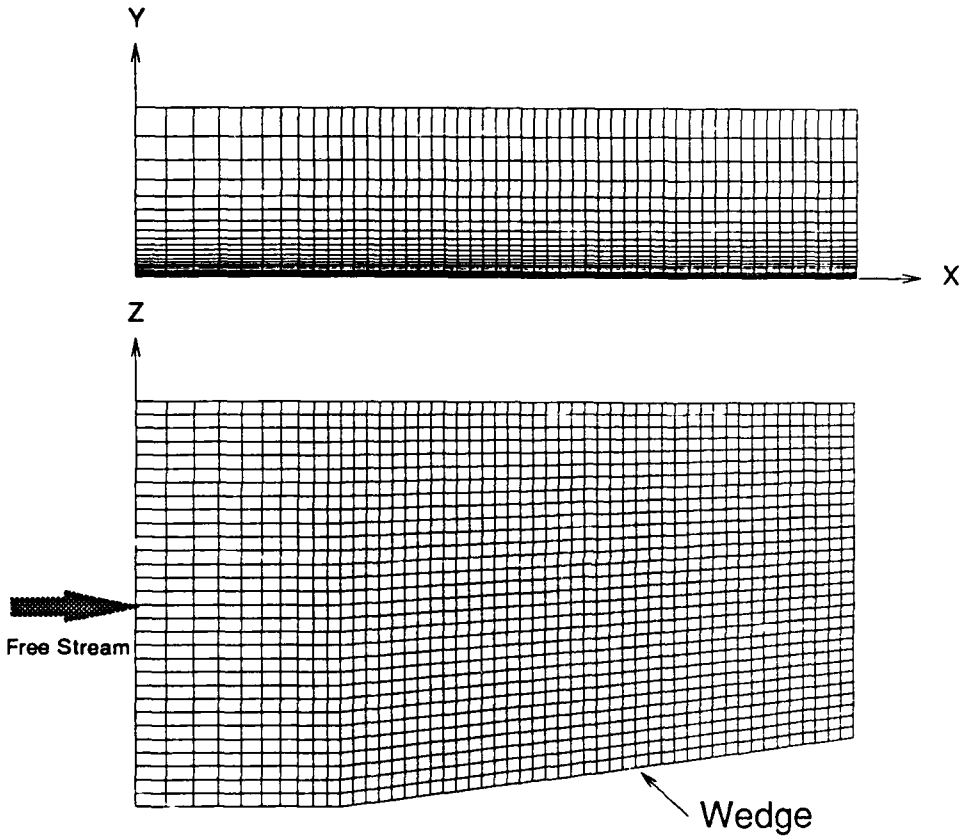


Figure 22: 50x30x30 computational mesh used to calculate the skewed shock/laminar boundary layer interaction flow field

The computational domain for the NS3D calculation was set up to capture a significant portion of the interaction region. The upstream boundary was positioned at the leading edge, 6 cm. upstream of the wedge. The downstream boundary was placed 15 cm from the wedge apex. The width of the domain was 5 cm, which was much greater than the thickness of the boundary layer on the plate. The height was 12 cm. The computational mesh, Figure 22, had 50 cells in the x-direction and 30 cells in the y- and z-directions for a total of 45,000 internal cells. This was not

an extremely fine mesh, but it did allow the dominant features of the flow to be modeled accurately. The 50 cells in the x-direction were divided such that 10 cells were stretched from the wedge apex to the leading edge of the plate and 40 cells were uniformly spaced along the wedge. The cells normal to the plate (y-direction) were stretched such that a minimum of 10 cells were in the boundary layer at the start of the interaction region. The z-direction mesh was uniform.

8.2 Results of NS3D Calculations

The NS3D calculation was performed on a CRAY2 computer, which provided the speed required to obtain a solution in a reasonable amount of time. The Encore computer would have required days of run time with eight CPU's to obtain a sufficiently converged solution on a mesh with 45,000 cells. Figure 23 provides the calculated pressure contours on the plate. The shock and upstream influence of the interaction

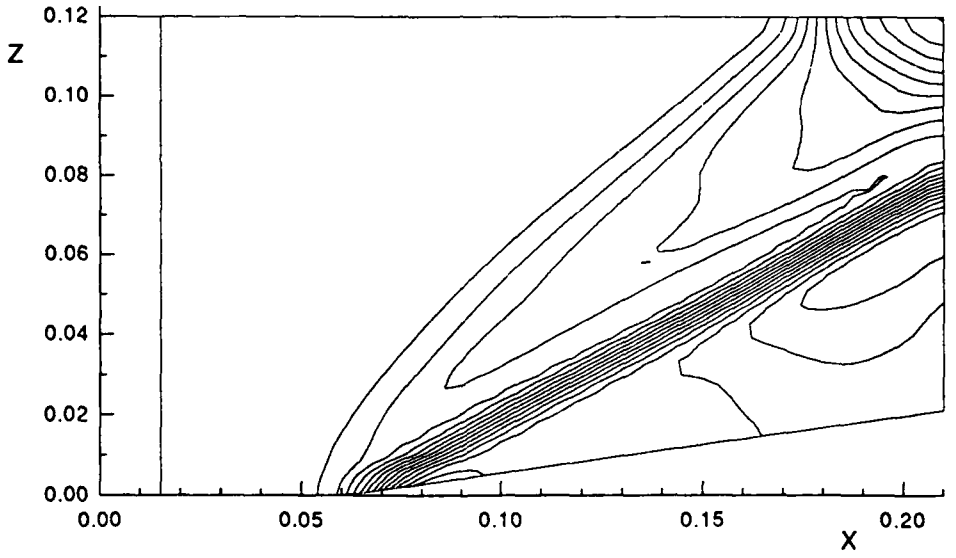


Figure 23: Surface pressure contours for the skewed shock/laminar boundary-layer interaction

region are easily identifiable. The calculated separation line is shown in Figure 24 which provides the velocity vectors in the cell next to the wall and several streamlines. The streamlines coalesce to highlight the separation line. The streamlines downstream of the separation line are evidence of the trapped vortex in the interaction region which sweeps the lower part of the boundary layer towards the separation

line. Surface streaklines obtained experimentally[40], Figure 25, show the same trends as the calculated streamlines. The calculated pressure rise along the surface in the

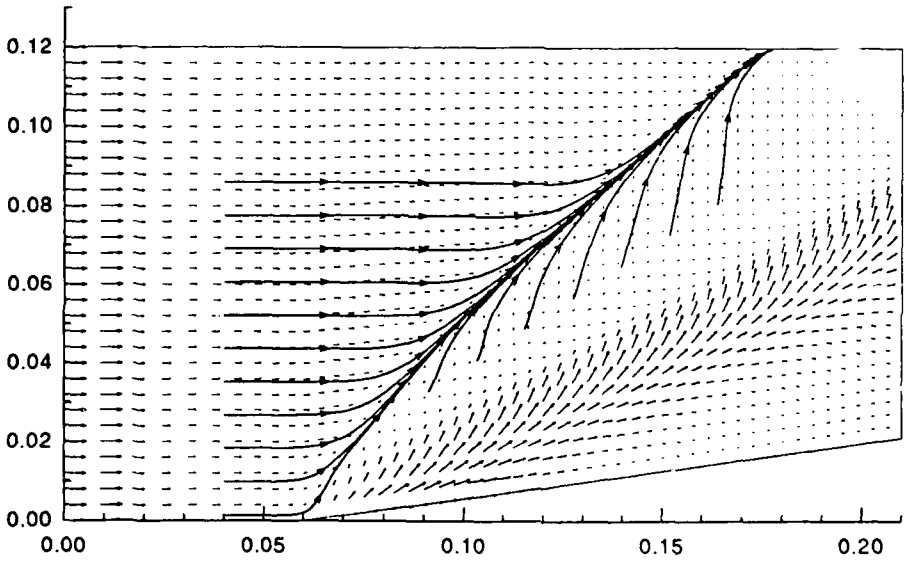


Figure 24: Velocity vectors and streamlines near the surface in the interaction region

Figure 25: Experimentally obtained surface streaklines

interaction region is compared with the experimental data in Figure 26. The initial

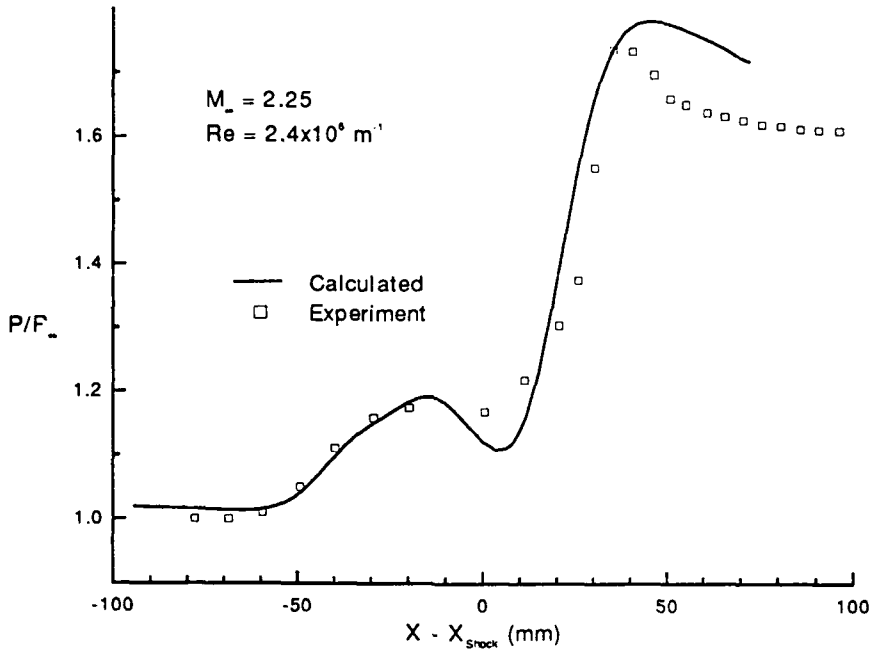


Figure 26: Surface pressure distribution at 5 cm above the wedge symmetry plane

pressure rise due to the upstream influence and the rise at the shock show good agreement. The calculated pressure distribution has a more pronounced dip upstream of the shock. The relaxation after the shock to the inviscid level does not show up in the calculated solution. This may have been due to the outflow boundary being too close to the shock. In all the NS3D calculation compares well with the experimental data.

8.3 Efficiencies of Parallel Algorithms

The effectiveness of the three parallel algorithms, which were implemented in the NS3D code, for complex 3D flow fields was evaluated by running the above test case on the Encore computer using a coarser mesh. For each algorithm the case was run nine times. The first run was made using one processor to provide a benchmark. Subsequent runs with a given algorithm were made with the number of processors incremented by one until nine processors were running in parallel. For each run the number of time steps was held fixed so that the computational work was equivalent. Since the speed of each processor on the Encore Multimax is 2–3 orders of magnitude

slower than the Cray 2, a computational mesh with 24x12x12 internal cells was used for these timing runs.

Each timing run was a two-step procedure. In the first step the computation was advanced for a specified number of time steps (50 for the explicit algorithm). For the second step the computation was rerun for just a few time steps (10 for the explicit algorithm). The run time for the second computation was then subtracted from the first to produce a net run time. The net run times were then used to calculate the relative efficiencies of the different parallel algorithms. The net run time removes the start-up time from the efficiency equation. The start-up consists of reading data and mesh files, which can not be parallelized. Of course, as the number of time steps becomes very large the start-up time becomes negligible in comparison to the total time, but this was not possible in view of the large number of runs.

Figure 27 summarizes the results of the timing runs for each algorithm. The net run time (elapse time) and efficiency are plotted versus the number of processors. Clearly, as the number of processors increases, the run times initially drop dramatically for all of the algorithms. The efficiencies of the algorithms are reasonably high for up to four processors. It is also seen that the run time tends to flatten out fairly quickly as the number of processors is increased. Of course this flattening must happen since the incremental effect of adding one more processor becomes small as the number of processors becomes large.

There are two other dominating effects that can be observed in the results. These are load balancing and the fraction of the code that is parallelized. The influence of poor load balancing is indicated most clearly by the sudden slope changes in the efficiency curves. When the number of processors does not divide evenly into the indexing range of the parallelized DO loops (i.e. the number of active cells in the index range), some of the processors will be idle for a portion of each cycle. Since there were 24 cells in the i-direction, the runs with 5, 7, and 9 processors should be affected by poor load balancing. This is readily apparent in the efficiency curves for the explicit and J-direction implicit algorithms, Figure 27. The Gauss-Seidel J-line implicit algorithm is even more strongly influenced by load balancing since it uses a three color {red - white - blue} scheme. Then the 24 must be divided by three. As a result only the runs with 2, 4, and 8 processors, which are factors of 8, will have good load balancing as demonstrated by the efficiency curve. It is important to note that as the ratio of the number of cells in the parallelized index direction to the number of processors become large the influence of load balancing becomes negligible since the idle time fraction becomes small.

The fraction of parallelized code, α, has a lasting influence that limits the effectiveness of a parallel algorithm. This is more clearly observed by plotting the

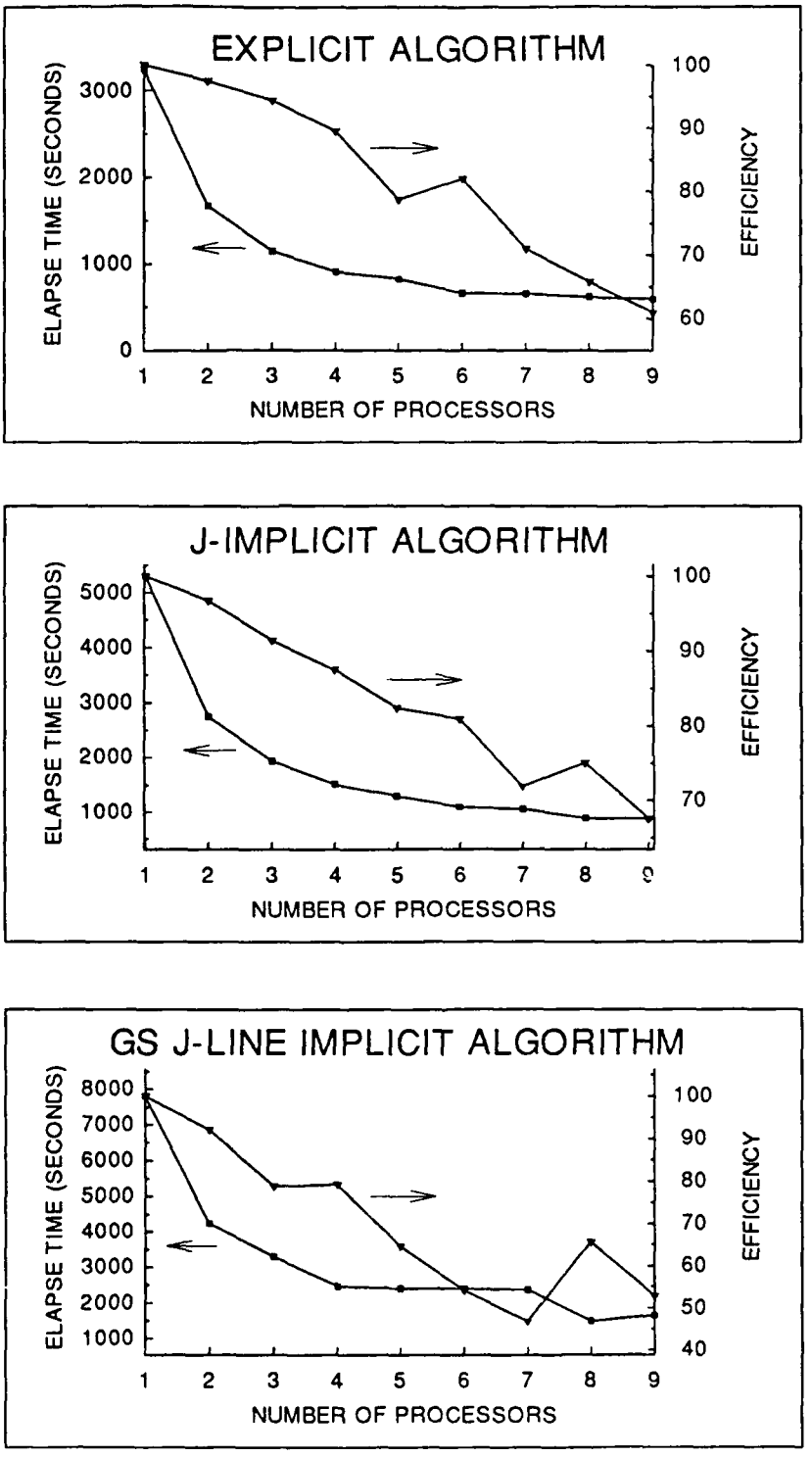


Figure 27: Run times and efficiencies versus number of processors for the three parallelized algorithms

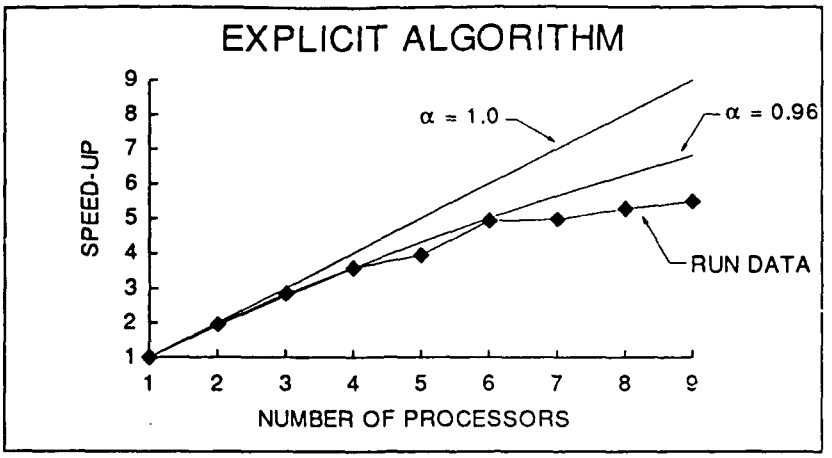
run data as speed-up versus number of processors, Figure 28. The speed-up is defined as the run time divided into the single processor run time. The $\alpha = 1.0$ curve represents the ideal 100% parallelized algorithm. The other α curves represent the speed-up calculated from Eqn. 7 using an α that best fits the run data. These curves lead to the conclusion that the explicit algorithm is 96% parallelized, the J-direction implicit algorithm is 95.3% parallelized and the Gauss-Seidel J-line implicit is 92% parallelized.

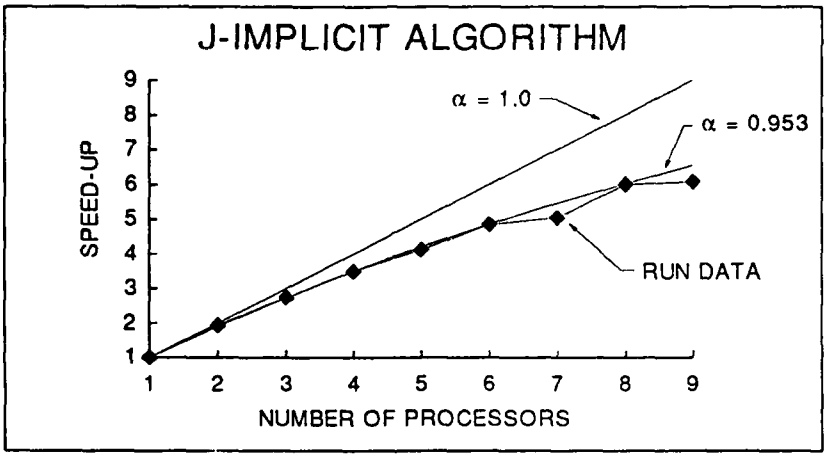
The data point for the 8 processor run with the explicit algorithm appears to be an irregularity since it should have fallen on the $\alpha = 0.96$ curve. However, as discussed in Section 7.2 the explicit algorithm was highly parallelized in the J-direction. Since 8 is not a factor of 12 (the number of interior cells in the J-direction), the load balancing was not really optimal for all of the DO loops for this run. Therefore, the data does exhibit the correct trend. The poor load balancing runs are otherwise very evident in Figure 28.

While the α's for the three parallelized algorithms appear to be very good, Amdahl's law says that as the number of processors becomes large, the sequential coding will ultimately limit the speedup. It is imperative that $\alpha \rightarrow 1$ if the benefits of the future MIMD machines with large numbers of processors are to be realized. This will be a primary concern during phase II of the research and development effort.

To demonstrate that the same trends apply to larger meshes, additional test case timing runs were made with the 50x30x30 mesh using the parallel J-direction implicit algorithm with 1,4,5,6, and 8 processors. The results of these runs are summarized in Figure 29. Up through six processors the efficiency of the algorithm is comparable to results obtained with the 24x12x12 mesh. The non-optimal load balancing does not significantly influence the 4 and 6 processor runs, which probably results from the small fraction of time that any processors are idle. The five processor run shows a definite improvement over the coarse mesh results. This was anticipated since 5 is a factor of 50 but not of 24. The eight processor run shows a strong influence of poor load balancing, which results in a substantial drop in efficiency. The speed-up curves confirm that 95 - 96% of the algorithm is parallelized as indicated by the coarse mesh results.

It was not possible to run any cases on the Encore Dual-Multmax LAmP computer, since a parallel fortran compiler was not available during the period of performance of the current contract. Had that computer been available, only one or two days of effort would have been required to complete the desired runs. However, the results obtained on our Encore Multimax computer using up to nine processors have provided sufficient data to satisfy our Phase I primary objective.





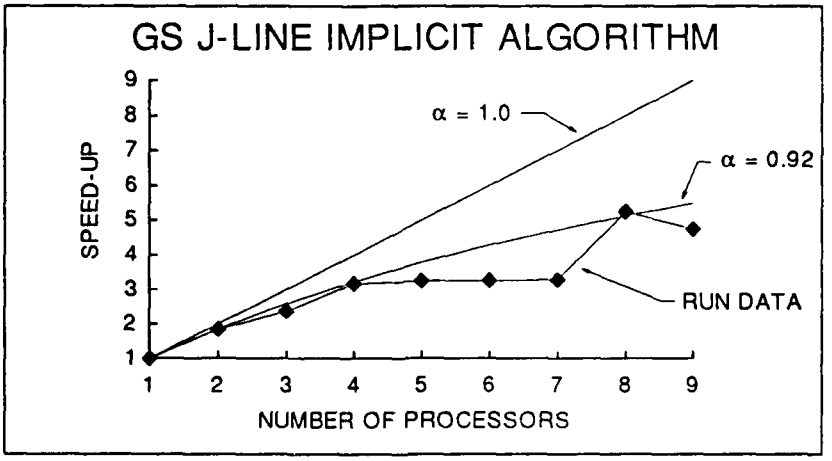


Figure 28: Run speed-up versus number of processors for the three parallelized algorithms

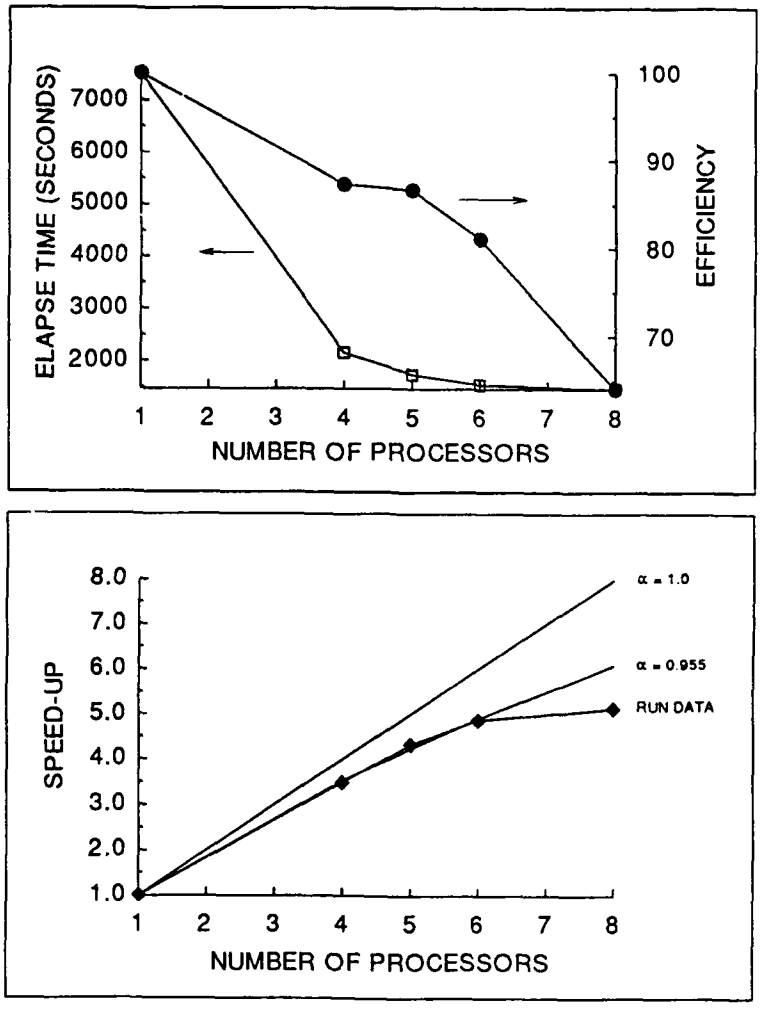


Figure 29: Run times, efficiencies, and run speed-up versus number of processors for the parallel J-direction implicit algorithm with the 50x30x30 mesh.

9 Conclusions

- Computational Fluid Dynamics is one area where parallel computing will be greatly beneficial.
- Using domain decomposition to create parallelism, a CFD code that solves the 3D Navier–Stokes equations was implemented on the Encore Multimax. Three different parallel algorithms were developed and benchmarked on a real engineering problem (one explicit and two implicit methods). The benchmark times indicate that between 92 and 96% of the code executed in parallel (the remaining portion executed sequentially). With up to 9 processors speedups for two different meshes closely follow Ware's relation, Equation 7. This indicates that inter-processor communication overhead was negligible. Variances of actual speedups from Ware's relation are explained by poor load balancing caused by "left over" tasks.
- We conclude from this preliminary work that by restructuring the NS3D code to minimize interprocess communication overhead and the fraction of serial computations, the NS3D code could achieve nearly 100% efficiency on the Encore Multimax and be very applicable to MIMD computers with large numbers of processors.
- As our experience and skills in parallelizing CFD algorithms increase, we expect that CFD software (like NS3D) can take maximum advantage of future MIMD computers with hundreds or more processors.
- Current "optimal" sequential CFD algorithms can be effectively parallelized. Research efforts should be directed in part at reordering and restructuring existing algorithms that are "optimal" on sequential computers.
- These results justify continuing the development of a 3D Navier–Stokes flow analysis code for parallel computers in Phase II.

References

[1] Peterson, V.L.,"The Impact of Supercomputers on the Aerospace Sciences," AIAA 24th Aerospace Sciences Meeting, Reno, NV, January 1986.

[2] Denning,P.J.,"Parallel Computation," *American Scientist*, Vol. 73, No. 4, July 1985, pp322-323.

[3] Moore,R., Nassi, I., O'Neil, J., and Siewiorek, D.P.,"The Encore Multimax: A Multiprocessor Computing Enviroment.", Encore Technical Report No. ETR 86-004, Encore Computer Corporation, Marlboro, MA, 1986.

[4] Nassi, I.,"A Preliminary Report on the Ultramax: A Massively Parallel Shared Memory Multiprocessor," Encore Technical Report No. ETR 87-004, 1987.

[5] Peery,K.M., Imlay,S.T., and Katsandres,J.T.,"Real Gas Blunt-Body Flow Simulations", AIAA Paper No. 87-2179, June 1987.

[6] Peery,K.M.,and Imlay,S.T. "An Efficient Implicit Method for Solving Viscous Multi-Stream Nozzle/Afterbody Flow Fields", AIAA Paper No. 86-1380, June 1986.

[7] Peery,K.M.,Ponten,B.D, and Roberts,D.W., "Simulation of Unsteady Two-Dimensional Inviscid Flow Fields Around Geometrically Complex Objects", AIAA Paper 85-1273, July 1985.

[8] Imlay, S.T.,"Numerical Solution of 2-D Thrust Reversing and Thrust Vectoring Nozzle Flowfields," AIAA Paper No. 86-0203, 1981.

[9] Imlay, S.T.,"A Solution Adaptive Grid/Navier-Stokes Solution Procedure," AIAA Paper No. 87-2180, June 1987.

[10] Imlay, S.T.,"Implicit Time-Marching Solution of the Navier-Stokes Equations for Thrust Reversing and Thrust-Vectoring Nozzle Flows," Ph.D. Dissertation in the Dept. of Aero. and Astro., Univ. of Wash., Washington, 1986.

[11] Anderson, D.A., Tannehill, J.C., and Pletcher, R.H., *Computational Fluid Mechanics and Heat Transfer*, McGraw-Hill, 1984

[12] Steger,J.L., and Warming,R.F., "Flux Vector Splitting of the Inviscid Gasdynamic Equations with Applications to Finite-Difference Methods," *Journal of Comp. Phys.*, Vol. 40, pp 263-29a3, 1981.

[13] Roe,P.L.,"Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp.357-372.

[14] Anderson, W.K., Thomas, J.L., and vanLeer, B., "A Comparison of Finite Volume Flux Vector Splitting for the Euler Equations," AIAA Paper No. 85-0122, 1985.

[15] Pulliam,T.H. and Steger,J.L.,"On Implicit Finite Difference Simulations of Three Dimensional Flows," AIAA Paper No. 78-10, Jan. 1978.

[16] Vinokur,M. and Liu,Y.,"Equilibrium Gas Flow Computations II: An Analysis of Numerical Formulations of Conservation Laws," AIAA Paper No. 88-0127, Jan. 1988.

[17] Cooper,J.R. and Hankey, W.L. Jr.,"Flowfield Measurements in an Axsymmetric Axial Corner at M=12.5," *AIAA Journal*, Vol.12, Oct. 1974, pp.1353-1357.

[18] Shang,J.S., and Hankey,W.L., "Numerical Solution of the Navier-Stokes Equations for a Three-Dimensional Corner," *AIAA Journal*, Vol.15, No.11, November, 1977, pp. 1575-1582.

[19] Mason, M.L., Putnam, L.E. and Re, R.J., "The Effect of Throat Contouring on Two-Dimensional Convergent-Divergent Nozzles at Static Conditions," NASA TP-1704, Aug. 1980.

[20] Johnson,G.M.,"Parallel Processing in Fluid Dynamics", ASME Fluids Engineering Conference, Cincinnati, Ohio, June14-18, 1987.

[21] Johnson,G.M.,et al.,"Multitasked Embedded Multigrid for Three-Dimensional Flow Simulation", Tenth International Conference on Numerical Methods in Fluid Dynamics", Beijing, China, june23-27, 1986.

[22] Johnson,G.M.,and Julie M. Swisshelm, "Multigrid for Parallel Processing Supercomputers", Third Copper Mountain Conference on Multigrid Methods, Copper Mountain, Colorado, April 6-10, 1987.

[23] Patel,N.R., Struek,W.B., and Jordan,H.F., "A Parallelized Solution for Incompressible Flow on a Multiprocessor", AIAA Paper No. 85-1511, July, 1985.

[24] Ortega, J.M. and Voight, R.G., *Solution of Partial Differential Equations on Vector and Parallel Computers*,SIAM, 1985.

[25] Evans, D.J., "Parallel S.O.R. iterative methods," *Parallel Computing*, Vol 1, 1984, pp3-18.

[26] Lomax, H. and Pulliam, T., "A Fully Implicit Factored Code for Computing Three-Dimensional Flows on the Illiac IV," *Parallel Computations*, Academic Press, NY, 1982.

[27] Eberhardt, S.D. and Baganoff, D., "Multiple Grid Problems on Concurrent Processing Computers," NASA TM 86675, Feb. 1986.

[28] Gibbons, A. and Wojciech Rytter, *Efficient Parallel Algorithms*, Cambridge Univ. Press, 1988.

[29] Patel, N.R. and Jordan, H.F., "A Parallelized Point Rowwise Successive Over-Relaxation Method on a Multiprocessor," *Parallel Computing, Vol 1*, pp207-222, 1984.

[30] Harding, A.F. and Carling, J.C., "The Three Dimensional Solution of the Equations of Flow and Heat Transfer in Glass Melting Tank Furnaces," *Supercomputer and Parallel Computation*, edited by Paddon, D.J., Calarendon Press, Oxford, 1984.

[31] Mandel, D.A. and Trease, H.E. , "Parallel Processing a Real Code—A Case History," LA-UR 88-1836, DE88-014456, 1988.

[32] Gropp, W. and Smith, E.B., "Computational Fluid Dynamics on Parallel Processors," Yale Univ. YALEU/DCS/RR-570, Dec 1987.

[33] Fox, G.C., "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems?," *Third Conf. on Hypercube Concurrent Computers and Applications*, Jet Propulsion Lab., Jan 19-20, 1988.

[34] Hiromoto, R.E., Lubeck, O.M., and Moore, J., "Experiences with the Denelcor HEP," *Parallel Computing*, Vol 1, pp197-206, 1984.

[35] Linden, J., Steckel, B., and Stuben, K., "Parallel Multigrid solution of the Navier-Stokes equations on general 2D domains," *Parallel Computing*, Vol 7, pp461-475, 1988.

[36] Fatoohi,R., "Multitasking on the Cray Y-MP," Report No. RNR-88-001, NAS Applied Research Office, NASA Ames Research Center, Dec. 1988.

[37] Schultz, M.H., Blackie, J.C., and McBurney, J.C., "Supercomputer Algotecture Analysis," *ESD: THE Electronic Sysetm Design Magazine*, pp53-57, April 1989.

[38] Catherasoo, C.J., AMETEK Computer Research Divison, Monrovia, California. Personal communication. April 1989.

[39] Keyes, D.E., "Domain Decomposition Methods for the Parallel Computation of Reacting Flows," NASA-CR-181719, Sep. 1988.

[40] Degrez,G. and Ginoux,J.J., "Surface Phenomena in a Three-Dimensional Skewed Shock Wave/Laminar Boundary-Layer Interaction," *AIAA Journal*, Vol.22, No. 12, December 1984, pp. 1764-1769.

[41] Ware, W., *The ultimate computer*, IEEE Spect., 10, 3, 1973, pp.89-91.